Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Aplicación Web de bases de datos en PHP usando el Framework Symfony

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Eloy José Gómez Tébar

Tutor: José Vicente Busquets Mataix

2014-2015

Resumen

Implementación de una aplicación web de consulta de registros de acceso para el administrador de un servidor de socios de un club de tenis, utilizando el conocido *framework* PHP, Symfony y las librerías Bootstrap y jQuery.

La aplicación permite buscar a cualquier socio y obtener todos sus registros de acceso, pudiendo además obtener gráficas de los datos obtenidos. Por otro lado, podrá acceder a la ficha del socio para modificar sus datos o para eliminarlos.

Además será posible realizar consultas usando como término de búsqueda una fecha, de esta forma podrá identificar usos irregulares del sistema más fácilmente.

Palabras clave: Symfony, registros de acceso, PHP, Aplicación Web.

Tabla de contenidos

1. Introducción

1 - Introducción

El objetivo de este proyecto es la implementación de una aplicación web de consulta de registros de acceso de un servidor de socios de un club de tenis. Esto permitirá al administrador obtener datos sobre la cantidad de veces que un socio ha accedido y el momento de dichas entradas. De esta forma, aparte de obtener listados de registros de acceso, también puede identificar usos irregulares y así evitar futuros problemas en el servidor.

Además, entre las capacidades del administrador estará la creación y eliminación de socios, así como la modificación de estos, dándole la oportunidad de tomar las medidas necesarias en caso de detectar un abuso o uso irregular. Así aumentamos la seguridad del servidor, identificando accesos no autorizados y actuando para evitar los siguientes.

Las herramientas usadas para hacer esta memoria han sido las siguientes:

- Microsoft Word
- LucidChart, una web muy útil que nos permitirá crear los diagramas de casos de uso, de clase y de actividad. (www.lucidchart.com)

2 - Symfony

Para realizar esta aplicación se ha hecho uso de Symfony[2]. Symfony es un *framework* PHP basado en el patrón Modelo-Vista-Controlador que permite optimizar el desarrollo de aplicaciones web.

Fue diseñado para facilitar y agilizar las tareas comunes de una aplicación web, de forma que el desarrollador pudiera centrarse en los aspectos específicos de su aplicación. Contiene una gran cantidad de clases orientadas a dicho fin, funciones y herramientas ya existentes, que Symfony nos ofrece en forma de librerías. De esta forma se atiende a la conocida premisa de "no reinventar la rueda"

Estas son algunas de las herramientas que permite usar Symfony:

- Uso de *templates* que permiten un sencillo diseño HTML sin necesidad de conocimientos de Symfony.
- El enrutamiento que permite crear rutas y asociarlas a una o varias acciones de un controlador.
- Una capa orientada a objetos que sustituye las funciones y variables globales de PHP.
- Dispone de un generador de formularios que permite crearlos y utilizar los datos obtenidos de una forma sencilla y automatizada mediante el uso de la capa orientada a objetos del punto anterior.
- Doctrine, una librería que simplifica la interacción con la base de datos, haciendo que en vez de trabajar con filas y columnas trabajes con objetos completos.

3 - Motivación y objetivos

La principal motivación u objetivo de la aplicación es permitir al administrador del servidor tener un mayor control sobre el acceso de los socios al servidor y así actuar para aumentar la seguridad en este.

La motivación personal a la hora de desarrollar este proyecto es aprender a programar en lenguaje PHP, así como afianzar conocimientos de HTML, CSS y JavaScript. También me parecía conveniente familiarizarme con el uso del *framework* Symfony, cada vez más presente en el desarrollo de aplicaciones web, mejorando además mis aptitudes hacia el esquema Modelo-Vista-Controlador.

4 - Entorno de desarrollo

Este trabajo se ha desarrollado en un entorno Ubuntu basado en Linux y puede ser utilizado en cualquier navegador, puesto que cumple los estándares básicos de estos.

El aspecto de la aplicación ha sido implementado usando el *framework* Bootstrap, por lo que la aplicación será totalmente adaptable a cualquier dispositivo.

2. Especificación de requisitos

1 - Introducción

Las siguientes definiciones y parámetros están redactados siguiendo el estándar IEEE-STD-830-1998.

1.1 - Propósito

Este apartado tiene como objetivo definir el comportamiento y la funcionalidad de la aplicación a desarrollar. Aquí hablaremos de la estructura de la aplicación y de los elementos creados, la función que tienen en la aplicación y la relación con el resto de elementos. Además también expondremos las repercusiones en la base de datos de las distintas tareas que realiza la aplicación.

Este texto va enfocado a usuarios con conocimientos básicos del lenguaje de programación PHP, así como de la programación orientada a objetos. Sin embargo, para mejorar la lectura y la comprensión de esta información, ha sido escrita de forma que alguien sin dichos conocimientos informáticos pueda hacerse una idea de los conceptos e ideas expresados.

1.2 - Alcance

Se desea realizar una aplicación llamada EntryLog usando una programación orientada a objetos, en PHP y utilizando el *framework* Symfony.

EntryLog tendrá como función hacer búsquedas de accesos hechos a un servidor de socios de un club de tenis. Estas búsquedas serán guardadas a modo de historial y podrán estar filtradas por socio o por fecha, dando como resultado un listado de entradas que también podrá ser imprimido en pdf o mostrado mediante estadísticas o gráficas. Por último, la aplicación permitirá al administrador crear socios y actuar sobre ellos ya sea modificando sus datos o eliminándolos.

El objetivo principal de EntryLog es controlar las entradas al servidor para así identificar usos irregulares y actuar para evitar futuros problemas.

1.3 - Definiciones, siglas y abreviaturas

PHP: *PHP Hypertext Pre-processor* es el lenguaje de programación que se ha utilizado para la realización del trabajo. Este lenguaje actúa del lado del servidor y fue diseñado principalmente para permitir el uso de contenido dinámico en las páginas web.

JavaScript: Es un lenguaje de programación utilizado principalmente para mejorar el dinamismo de las páginas web.

Framework: Es un esquema o patrón que define la estructura de un proyecto así como el modelo de comportamiento entre los elementos que los componen.

Symfony: Este es el *framework* sobre el que desarrollaremos nuestra aplicación.

Aplicación web: Una aplicación cuyo uso requiere el acceso a un servidor web. Una de las características de estas aplicaciones es que no se limitan a mostrar información, realizan tareas que en algunas ocasiones incluyen el uso de bases de datos.

MVC: Modelo-Vista-Controlador es el patrón de arquitectura de software usado para la creación de esta aplicación. Su principal característica es que separa los datos y la lógica de negocio de la interfaz.

Middleware: Es un software que sirve para el intercambio de información entre aplicaciones o entre capas pertenecientes a una misma aplicación. Es, por así decirlo, una especie de intermediario.

1.4 - Referencias

-IEEE-STD-830-1998
-Wikipedia
-http://es.slideshare.net/EdgarDueas/symfony-2-27787361
-http://librosweb.es/libro/symfony_2_4
-www.symfony.es

1.5 - Visión global

Esta es la estructura básica que sigue esta ERS hasta ahora. Seguidamente exponemos una descripción general de la aplicación así como sus funcionalidades y los elementos que la componen.

2 - Descripción general

Esta aplicación web deberá permitir al administrador de un servidor, obtener los registros de acceso a este, pudiendo filtrar los datos por socio o por fecha.

Estos datos deberán poder mostrarse en listados, estadísticas o gráficas, además de ser imprimidos en formato *pdf*. Por último también deberá dar la posibilidad al administrador de crear socios, modificar sus datos y eliminarlos.

3 - Requisitos específicos

3.1 - Interfaces externas

Las interfaces mostradas al usuario, es decir, al administrador intentaremos que sean intuitivas, fáciles de usar y entender. Pensamos que esta es una tarea fácil debido a que el administrador está bastante acostumbrado a interfaces gráficas como la desarrollada.

En este trabajo no hemos configurado ninguna relación con otros sistemas, por tanto, no hemos tenido que preocuparnos por las interfaces de hardware.

3.2 - Funciones

Aquí hablaremos del comportamiento de la aplicación y de las funcionalidades que ofrece y de las tareas que realiza:

3.2.1 – Clase Socio

Esta clase o entidad será la que englobe las propiedades de los usuarios y será sobre la que el administrador tendrá más capacidades, pues podrá crear, modificar y eliminar los objetos de esta clase. He aquí su composición:

Nombre	Tipo	Consideraciones
id	integer	Único, clave primaria.
pass	string	Más de 5 caracteres y menos de 15.
nombre	string	No debe contener números o signos de puntuación.
apellido1	string	No puede ser null.
apellido2	string	
email	string	Debe tener el formato de una dirección de email.
fechaNacimiento	datetime	No puede ser null.

3.2.1.1 – Crear Socio

Como nuestra aplicación solo será desarrollada en el *backend* tenemos que encontrar una manera de poder crear socios desde este. Por esta razón el administrador tendrá la posibilidad de crear socios, pudiendo introducir todos y cada uno de los datos. Estará compuesto por la información básica de cualquier persona, siendo fácilmente extensible en caso necesario:

Acción	Repercusión en la base de datos
Acceder a la página de socios	Se hará una consulta para obtener los socios ya existentes.
Pinchar en el enlace de “Crear socio”	
Rellenar el formulario y enviar	Correcto: El objeto Socio será rellenado con los datos introducidos e insertado en la tabla que corresponda a la clase Socio de la base de datos. Error: Los formularios de Symfony pueden mostrar los mensajes de error, sin necesidad de desarrollar un mensaje. No tendría ningún efecto.
Redirigir a la página de socios	Volverá a consultar la base de datos para mostrar los socios, incluyendo al recién creado.

3.2.1.2 – Modificar Socio

Esta es la función que el administrador podrá hacer tanto desde la página de socios como desde los listados de registros. En este último caso solo podrá modificar los datos de los socios que aparezcan en alguno de los registros:

Acción	Repercusión en la base de datos
Acceder a la página de socios o al listado	Página de socios: Se hará una consulta para obtener los socios ya existentes.
Pinchar en el enlace de "Modificar socio"	
Rellenar el formulario y enviar	Correcto: El objeto Socio será rellenado con los datos introducidos y actualizado en la tabla que corresponda a la clase Socio de la base de datos. Error: Los formularios de Symfony pueden mostrar los mensajes de error, sin necesidad de desarrollar un mensaje. No tendría ningún efecto.
Redirigir a la página de socios	Volverá a consultar la base de datos para mostrar los socios, teniendo en cuenta las modificaciones realizadas.

3.2.1.3 – Eliminar Socio

Otra de las capacidades que tendrá el administrador es la de eliminar a un socio si ve un comportamiento irregular por parte de este. Este será el proceso:

Acción	Repercusión en la base de datos
Acceder a la página de socios o al listado	Página de socios: Se hará una consulta para obtener los socios ya existentes.
Pinchar en el enlace de "Eliminar socio"	Se eliminará al socio de la tabla correspondiente a Socios de la base de datos.
Redirigir a la página de socios	Volverá a consultar la base de datos para mostrar los socios, entre los cuales ya no estará el eliminado.

3.2.2 – Clase Búsqueda

En Symfony la recogida de datos de un formulario no se hace a través de las variables globales como $_POST y $_GET, se hace asignando al formulario un objeto, que se llenará con los datos recogidos.

Para ello hemos creado la entidad o clase Búsqueda, que recogerá los términos de búsqueda que inserte el administrador para filtrar los registros. Contendrá los dos posibles términos de búsqueda, usuario y fecha:

Nombre	Tipo	Consideraciones
id	integer	Único, clave primaria
usuario	string	
fecha	datetime	Año, mes y día deben estar todos rellenos o todos vacíos.

3.2.2.1 – Guardar búsqueda

Con el fin de añadirle una funcionalidad más avanzada a la clase búsqueda, hemos creado una tabla en la base de datos donde se guarden, algo así como un historial de búsquedas. A secuencia sería la siguiente:

Acción	Repercusión en la base de datos
Acceder a la página de búsqueda o a la de listado, en ambas se puede buscar.	
Rellenar los términos de búsqueda y procesar formulario.	Correcto: Se guarda el objeto rellenado en la tabla correspondiente a las búsquedas. Error: Los formularios de Symfony pueden mostrar los mensajes de error, sin necesidad de desarrollar un mensaje. No tendría ningún efecto.
Redirección al listado de registros	

3.2.2.2 – Borrar historial de búsquedas

El administrador tendrá la opción de borrar el historial de búsquedas. Consiste en lo siguiente:

Acción	Repercusión en la base de datos
Acceder a la página de búsqueda.	
Pinchar en el botón de borrar historial.	Se borraran todas las filas de la tabla de la base de datos correspondiente a las búsquedas.
Redirigir a la página de búsqueda.	

A modo de seguridad no se podrán eliminar filas separadas, pues cualquiera podría obtener información de socios y borrar su rastro, como si no hubiera pasado nada. De esta manera, aunque se puede borrar el registro entero, la ausencia del resto será un indicador claro de intrusión.

3.2.3 – Clase Registro

Esta clase ha sido creada para definir la estructura de los registros mostrados en los listados. A la hora de pasar los registros encontrados al *template* se podrían haber pasado listas (*arrays*) que contuvieran otras listas, pero hemos optado por pasar listas de registros. De esta manera hacemos una aplicación más robusta y más afín a la programación orientada a objetos.

Además esta clase ayudará a futuros desarrollos en los que se guarden los registros en bases de datos o se utilicen como atributos de otras clases. Un registro contendrá la *url* accedida, el *controller* accionado, el usuario (nombre e id de socio) y el momento en que se realizó el acceso:

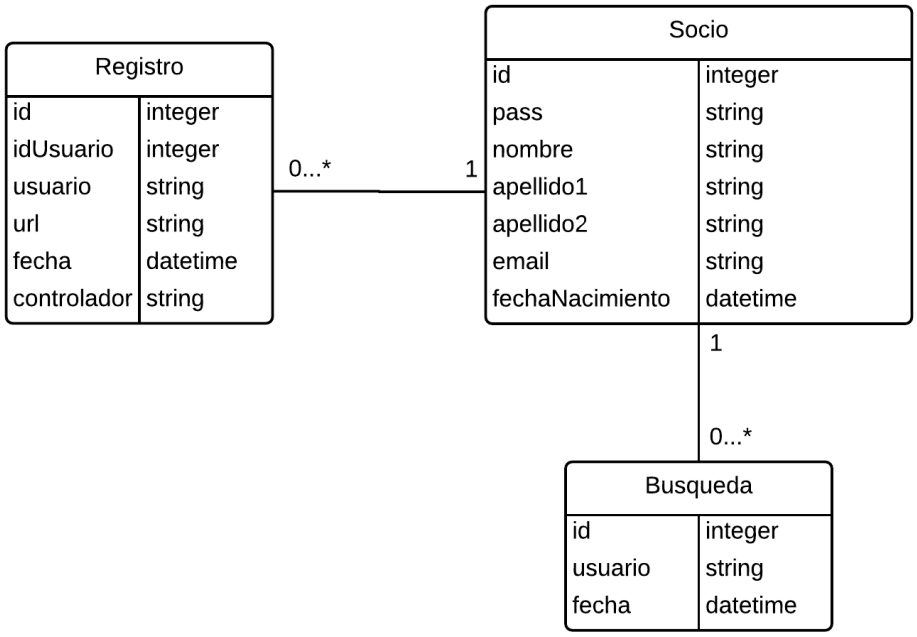
Nombre	Tipo	Consideraciones
id	integer	Único, Clave primaria.
idUsuario	integer	
usuario	string	
fecha	datetime	Aquí guardaremos tanto la fecha como la hora.
controlador	string	
url	string	

3. Análisis

1 - Introducción

Aquí se ofrece un análisis de la aplicación en forma de diagramas que mostraran las distintas funcionalidades de la aplicación y las relaciones que unen cada uno de sus elementos.

2 - Diagrama de clases

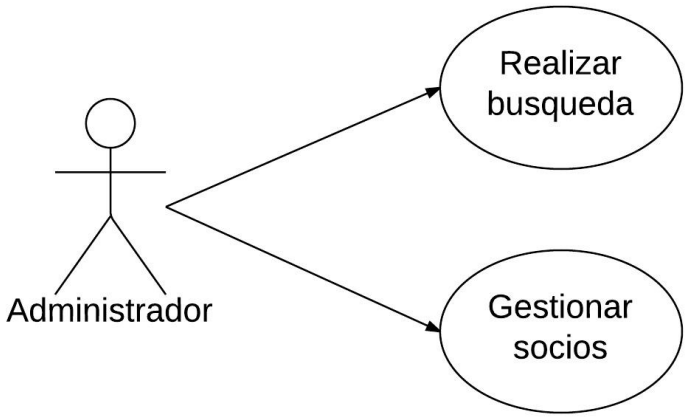


3 - Diagramas de casos de uso

Estos diagramas mostraran las diversas tareas que pueden realizar los usuarios de la aplicación y las tareas que podrán realizar en el estado en que desemboquen. En nuestro caso solo tenemos un tipo de usuario, que será el de administrador.

3.1 - Caso base

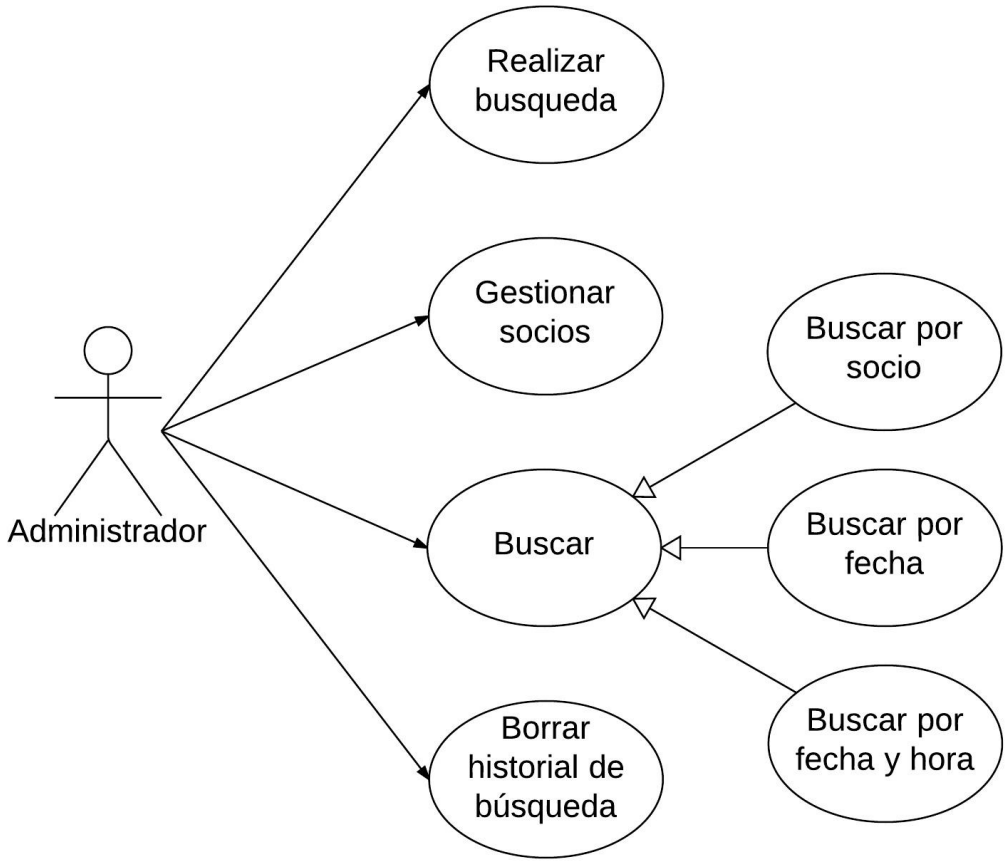
Este será el estado desde el que empezará la aplicación y consistirá en una página de bienvenida desde donde podremos acceder a las distintas secciones de la aplicación mediante un menú horizontal, situado en lo alto de la página, que estará presente en todas las páginas de la aplicación.



3.2 – Realizar búsqueda

En esta sección podemos realizar una búsqueda introduciendo dos tipos de términos: socio y fecha. La búsqueda podrá hacerse usando cualquiera de los dos tipos o ambos a la vez.

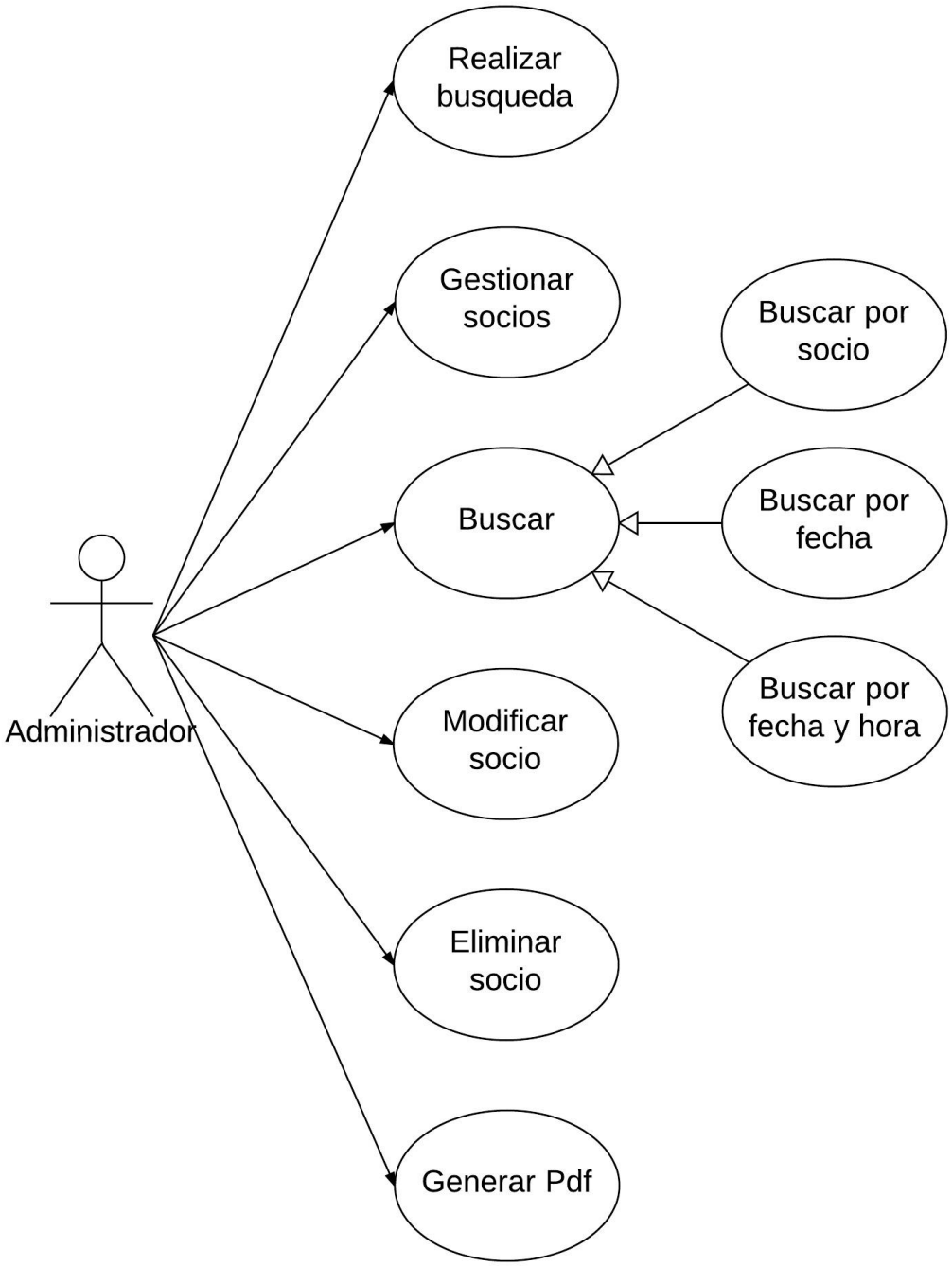
También estará presente el menú horizontal, en este caso la opción "Realizar búsqueda" únicamente recargará la página que ya estamos viendo.



3.3 - Listado

Aquí vemos las opciones que tendrá el administrador una vez ha realizado la búsqueda. En este punto, al administrador se le habrá mostrado un listado de registros de entrada, sobre ellos podrá generar un Pdf, modificar datos del socio al que pertenezca un registro en particular o eliminar dicho socio.

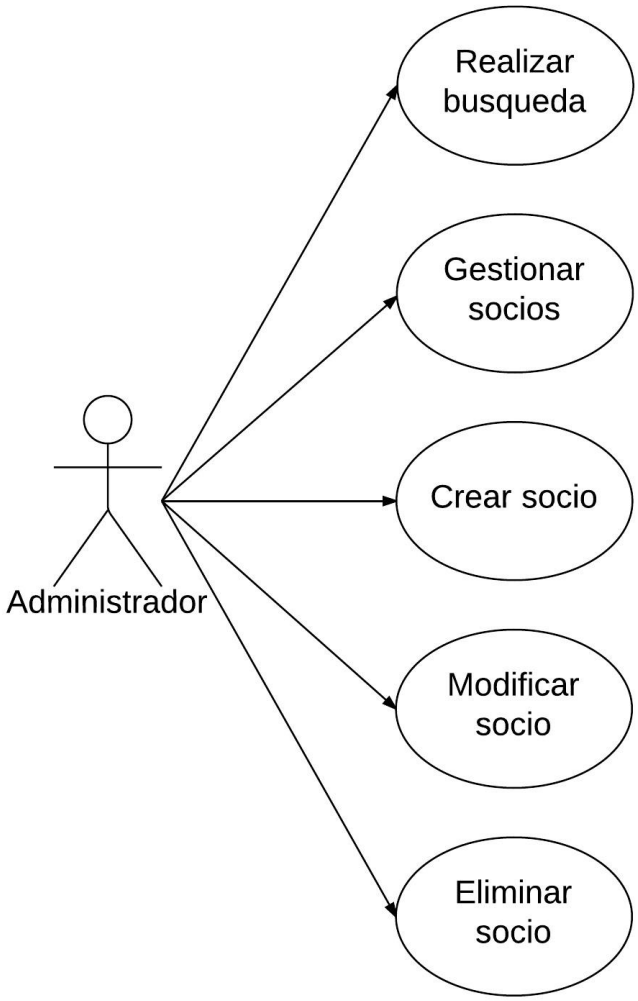
Además también podrá realizar otra búsqueda del mismo modo que en el caso base, con lo que evitamos que tenga que volver atrás para realizar una nueva consulta. Sin embargo, si se quiere ver el historial de búsquedas habrá que ir a "Realizar búsqueda" del menú.



3.3 – Gestionar socios

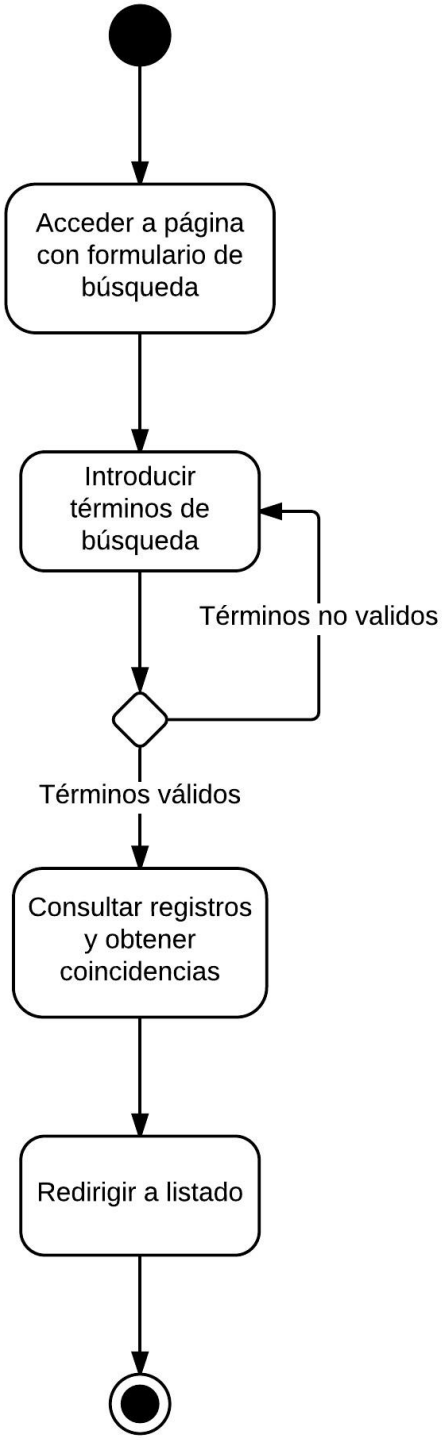
En este punto, el administrador estará viendo una tabla con los nombres de los socios existentes, sobre los cuales podrá actuar para modificar o eliminar sus datos. Además también podrá crear socios para realizar pruebas o rehacer un socio existente, eliminándolo y luego creándolo.

Como es normal también veremos el menú horizontal con sus opciones, entre las que se incluirá la opción “Gestionar socios”, lo cual recargará la página así como su información.

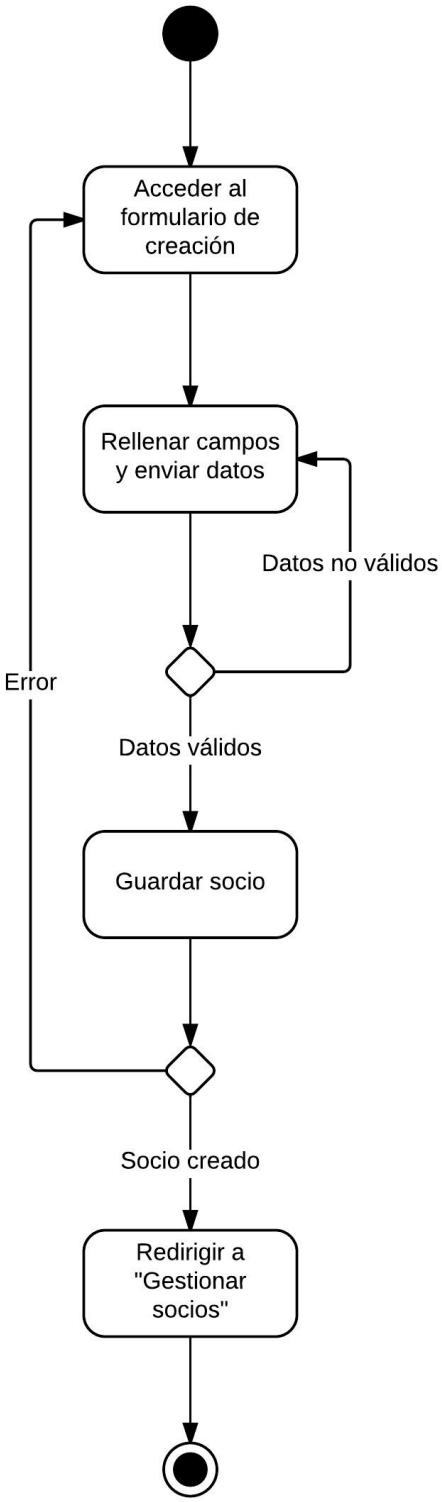


4 - Diagramas de actividad

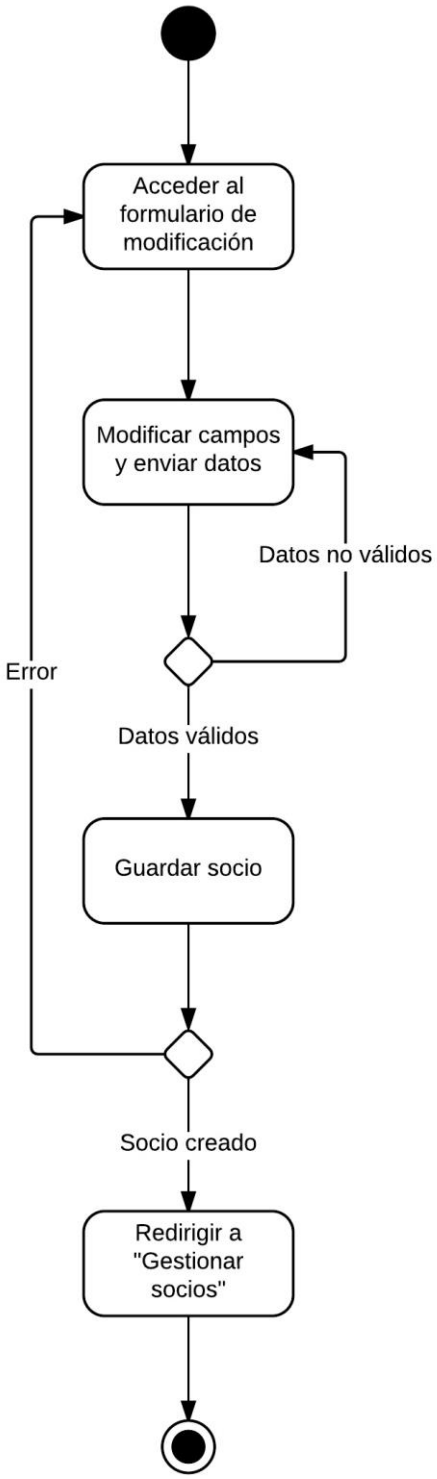
4.1 - Realizar búsqueda



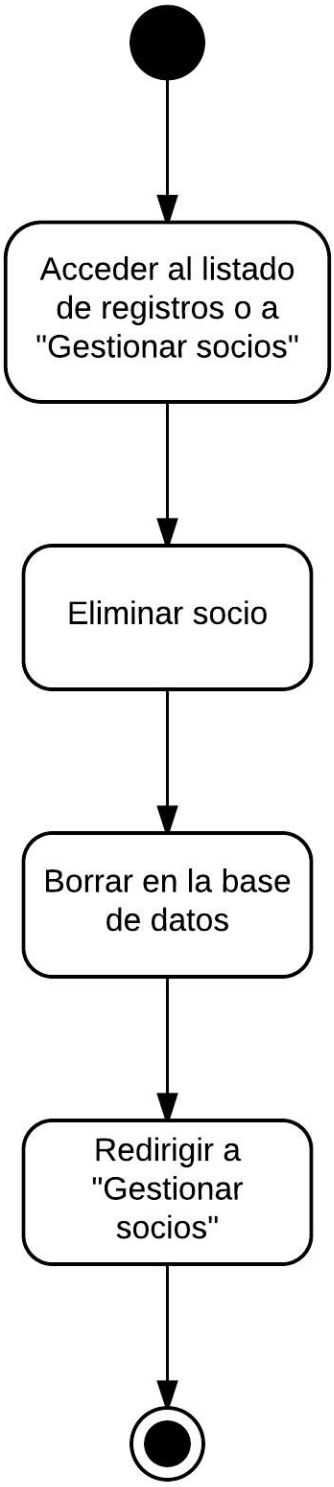
4.2 - Crear socio



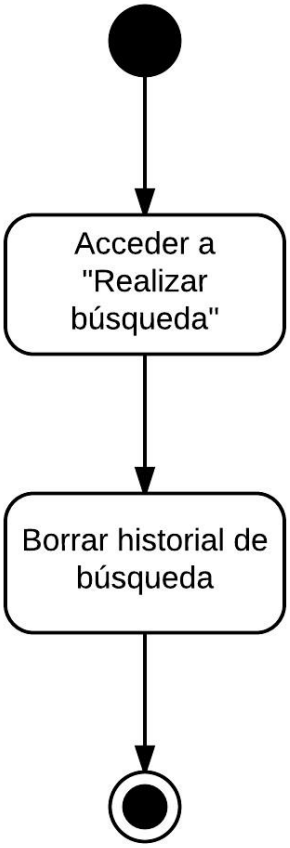
4.3 - Modificar socio



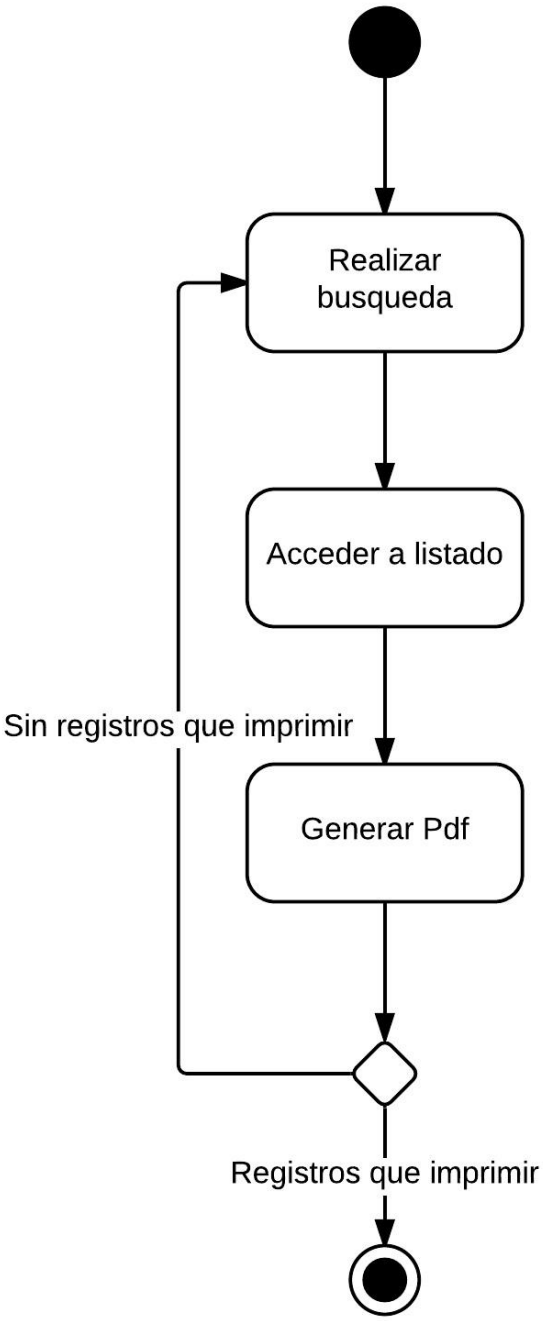
4.4 - Eliminar socio



4.5 - Borrar historial de búsquedas



4.5 – Generar Pdf



4. Diseño

1 - Introducción

Este apartado está dedicado a exponer los distintos esquemas y patrones seguidos para diseñar la aplicación. Explicaremos los motivos para su elección, así como las características que lo definen.

2 - Patrón de diseño

El patrón usado a la hora que desarrollar EntryLog es el Modelo-Vista-Controlador, el cual separa en tres capas la estructura de la aplicación. La elección de este esquema fuertemente unida a la elección del mismo trabajo, puesto que Symfony trabaja con este patrón.

Symfony está desarrollado principalmente para seguir este patrón:

- La capa **Modelo** o de datos está constituida por la información de la base de datos y por la forma en que la manejamos.

- La capa **Vista** se refiere a los archivos TWIG, muy parecidos a los archivos HTML, que definen las representaciones visuales de los datos que serán devueltas por el controlador.

- La última capa es la de **Controlador**, que será la que responda a los distintos eventos y la que, en esencia, actuará de *middleware* entre las capas de Modelo y Vista.

A continuación procederemos a detallar las características y comportamientos de cada una de las capas usando términos propios de Symfony, pero también de EntryLog para mejorar la comprensión.

2.1 - Modelo

En Symfony esta capa está definida tanto en la base de datos como en las clases creadas, las cuales se mantienen en constante relación, puesto que por cada clase creada por el desarrollador también se definirá una nueva tabla en la base de datos, la cual incluirá los atributos de dicha clase como campos de la tabla.

De esta manera la información de los objetos de una determinada clase se guardaran en la tabla asociada a dicha clase. Para ello lo único que tenemos que hacer es crear el objeto, dando valor a sus atributos, y luego *persistirlo*, es decir, guardarlo, para lo cual no es necesario ejecutar ninguna función SQL, pues Symfony nos ofrece sus funciones PHP.

Estas funciones forman parte de la librería Doctrine, que nos ayudará a manejar la información de nuestra base de datos de una manera mucha más intuitiva y simplificada. No trabajaremos sobre tablas, sino que trabajaremos directamente con objetos.

Como vemos, durante todo este proceso no han intervenido ninguna de las otras dos capas, poniendo así de manifiesto la independencia entre estas y las ventajas de utilizar este patrón.

En nuestra aplicación está capa contendrá las siguientes clases:

- Búsqueda: Guardará información sobre las búsquedas hechas anteriormente a modo de historial.

- Registro: Contendrá la información extraída de los registros. De esta clase no guardaremos objetos, pero la usaremos para estructurar la información mostrada en los listados.

- Socio: Aquí tendremos los datos de los socios y serán los únicos objetos sobre los que el administrador tendrá control total.

2.2 - Vista

Esta capa está constituida por los archivos visuales, la interfaz sobre la que actuará el usuario, en nuestro caso el administrador, provocando los eventos que más tarde el controlador manejará.

Estos archivos, en Symfony, son de tipo TWIG, es decir, unas plantillas que usan tanto el aspecto de los archivos HTML como la funcionalidad ofrecida por PHP. Además estas plantillas pueden heredar unas de otras, es decir, puedes crear una plantilla que solo contenga los elementos presentes en toda la aplicación (menú horizontal, logo, cabecera, pie de página…) y crear las demás plantillas heredando de esta y añadiendo los datos del cuerpo que deseemos.

En nuestro caso nuestro plantilla base contiene un menú horizontal que permitirá la navegación a las distintas secciones de la web. Como hemos dicho, este menú estará presente en todas las páginas de la aplicación web.

El resto de plantillas creadas son las siguientes:

- La interfaz de la página inicial, la cual solo contendrá un cuadro de bienvenida con una descripción de la aplicación.

- El formulario de búsqueda, el cual nos ofrecerá los campos de usuario y fecha para filtrar los futuros resultados.

- El listado de registros mostrados, el cual incluirá estadísticas o gráficas dependiendo de las acciones del administrador.

- La página de gestión de socios que contendrá un listado de socios, así como las posibles acciones que puedan realizarse sobre estos.

- Formulario de registro/modificación de socio.

2.3 - Controlador

Esta es la capa que responderá a los eventos producidos por el usuario en la capa de Vista, realizando peticiones a la capa de Modelo para enviar de nuevo información a la capa de Vista y que esta pueda mostrarla.

Symfony identifica estos controladores en archivos PHP, los cuales toman información también de las rutas del navegador, es decir, de la dirección URL. Esto es lo que Symfony define como *routing*.

El *routing* permite definir rutas y relacionarlas con funciones dentro de los controladores, que para Symfony serán *actions*. Estas rutas también pueden incluir variables, las cuales podremos tomar en dichas funciones, es decir, podemos definir una única ruta con una variable que será enviada al controlador, el cual tomará esa variable y actuará en consecuencia a ella.

En nuestra aplicación esto será muy útil para la página de creación y modificación de socios, pues podremos definir una sola ruta para crear y otra para modificar todos los socios. Además también definiremos la navegación de la web mediante el enrutamiento.

3 - Funcionamiento general de una aplicación MVC

En este tipo de aplicaciones, el usuario accederá a la ruta de la página de inicio de la aplicación, lo que provocará que el controlador lea dicha ruta y ejecute la función *action* asociada a esta. Esta función podrá contener peticiones tanto a la vista (variables en la ruta), como al modelo (base de datos), pero lo que seguro hará es devolver como respuesta una nueva vista.

El controlador seguirá controlando la ruta introducida, pero también actuará en cuanto el usuario realice una acción sobre la interfaz presentada, ejecutando de nueva la función que le corresponda y devolviendo una vista.

Como vemos, después ejecutar la función se volverá al punto inicial y el proceso se repetirá tantas veces como sea requerido.

4 - Ventajas

- Al tener separadas los datos de la interfaz, se puede dividir el trabajo de una mejor manera, pues alguien que no sepa nada de bases de datos ni de controladores puede diseñar la interfaz y viceversa.
- Las aplicaciones que usan MVC ofrecen una gran escalabilidad así como un fácil mantenimiento.
- Las modificaciones que se realicen en una de las capas no afecta en gran medida a las demás.
- Fomenta la reutilización de código.

5 - Inconvenientes

- El inicio de un proyecto basado en MVC requiere más tiempo que con otros patrones.
- La curva de aprendizaje es mayor que la de otros modelos.
- Adoptar una estructura predefinida que puede resultar perjudicial a la hora de desarrollar un tipo concreto de aplicación.
- Es necesario crear y mantener más ficheros.
- En el caso de que se utilice un lenguaje que no esté orientado a objetos la implementación es mucho más complicada ya que MVC se basa precisamente en este paradigma.

5. Implementación

1 - Instalación y configuración para desarrollos en un entorno Symfony

1.1 - Requisitos previos

Antes de instalar y configurar Symfony es necesario que tengamos instalados un servidor web con PHP y un servidor de base de datos SQL. En este caso, al haber realizado la aplicación en Ubuntu, utilizaremos Xampp como servidor Apache y phpMyAdmin como gestor de base de datos.

Para instalar Symfony usaremos el método que aparece en la página oficial de Symfony (symfony.com). A continuación mostraremos paso a paso cómo se instala y se inicia el desarrollo de una aplicación en Symfony.

En primer lugar instalamos Symfony mediante un comando cURL, una herramienta de transferencia de archivos que usaremos para descargar y ejecutar el instalador. En caso de que nuestra instalación PHP no disponga de cURL lo instalaremos con el siguiente comando:

```
sudo apt-get install php5-curl
```

1.2 - Instalación

Una vez instalado cURL ya podemos ejecutar el comando que nos facilita la web de Symfony.com:

```
curl -LsS http://symfony.com/installer -o /usr/local/bin/symfony
```

Sin embargo con instalarlo no basta, también deberemos darle los permisos necesarios para poder utilizarlo:

```
chmod a+x /usr/local/bin/symphony
```

Con esto ya podremos crear proyectos Symfony. Para ello usaremos la terminal para ir a la carpeta htdocs del servidor web y crear el proyecto en dicha carpeta:

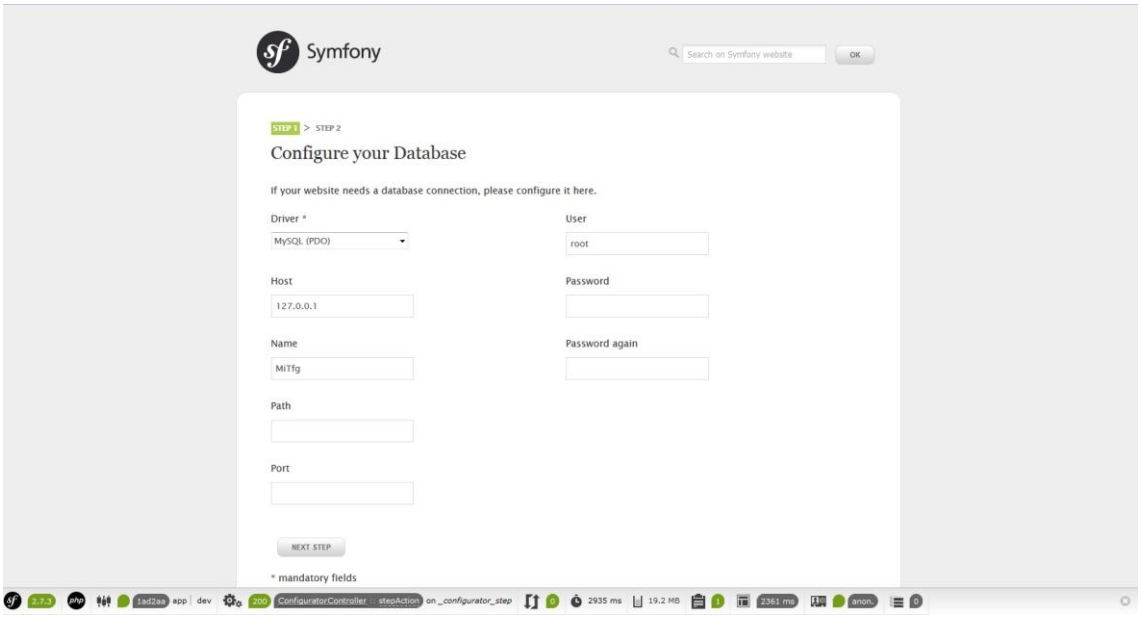
```
symfony new EntryLog
```

Tras ejecutar este comando veremos cómo se ha creado toda la estructura de ficheros y directorios, compuesta por las librerías y herramientas que usaremos.

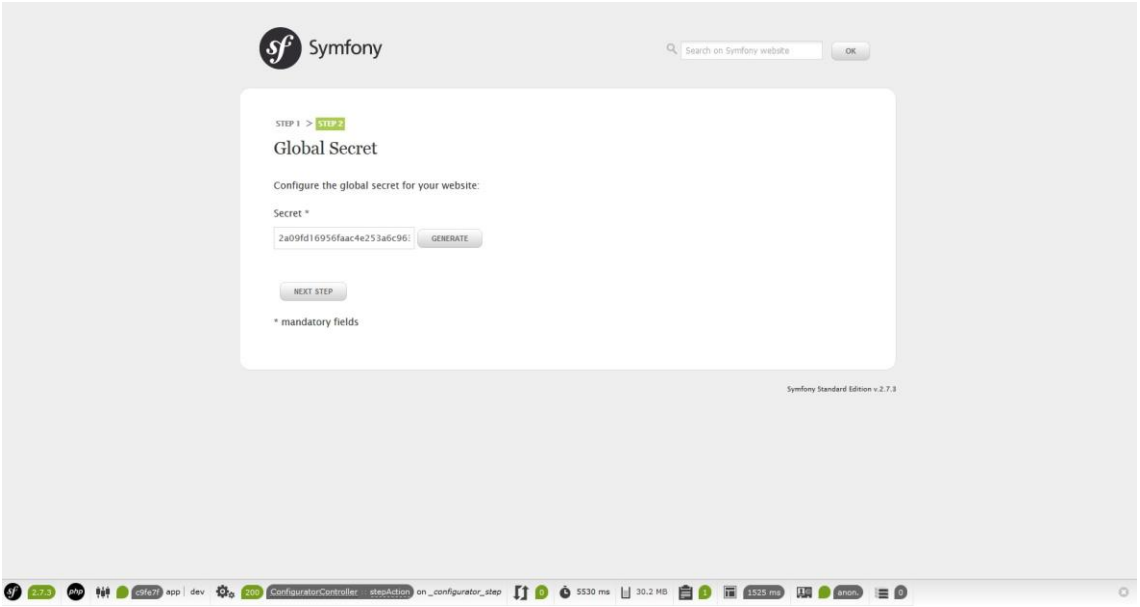
1.3 – Configuración de la base de datos

Para completar la instalación base del proyecto Symfony procederemos a asociar al proyecto una base de datos. Hay dos formas de hacerlo, la primera es bastante más automática y consiste en configurarla mediante la interfaz de Symfony, es decir, accediendo a la página principal de Symfony y desde ahí a la configuración del servidor (DOMINIO/config.php), de forma fácil e intuitiva:

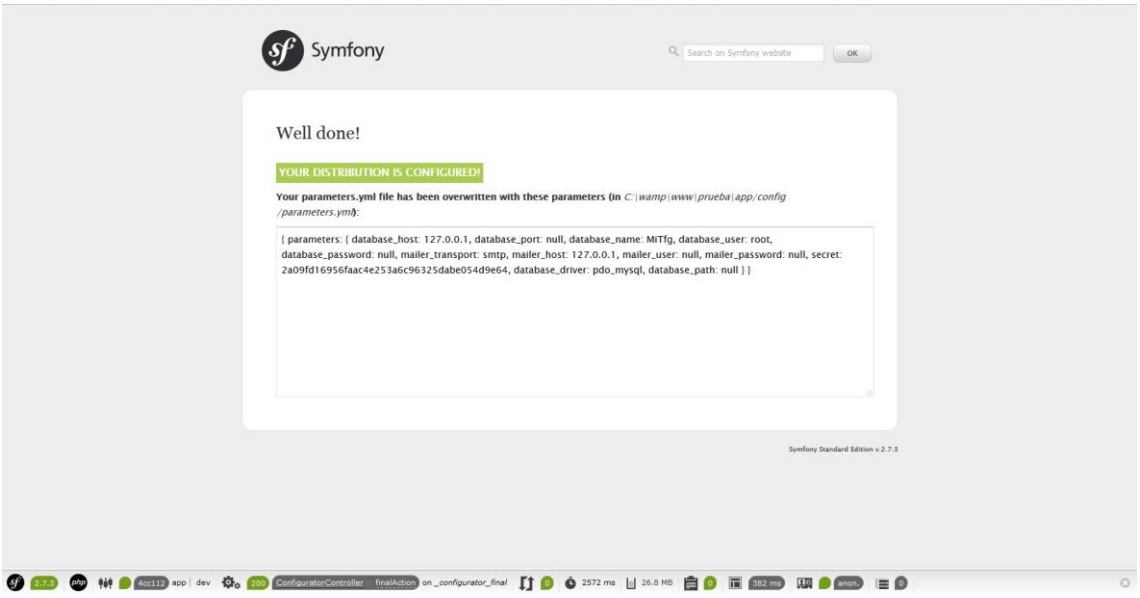
Configuración (config.php)



Al darle a siguiente (Next_step) nos permitirá crear una especie de "clave global secreta" que se utilizará en algunas partes del proyecto para evitar que un usuario malintencionado provoque que otro usuario envié información que no quiere enviar. Esta protección es conocida por Symfony como CSRF.



Cuando terminemos con la configuración nos informará de que el archivo parameters.yml ha sido modificado para incluir la información de acceso de nuestra base de datos.



Precisamente la segunda forma de configurarla es modificando este archivo manualmente:

```
# This file is auto-generated during the composer install
parameters:
    database_host: 127.0.0.1
    database_port: null
    database_name: entryLog
    database_user: user
    database_password: password
    mailer_transport: smtp
    mailer_host: localhost
    mailer_user: null
    mailer_password: null
    secret: global_secret
```

Finalmente, tras realizar correctamente la configuración de una forma u otra, introduciremos el siguiente comando para crear la base de datos:

```
php app/console doctrine:database:create
```

Ahora si accedemos a DOMINIO/phpmyadmin veremos que la base de datos ha sido creada.

2 - Bundles

2.1 - Introducción

En Symfony los bundles son directorios que contienen los ficheros y directorios necesarios para realizar una función específica, desde controladores y vistas hasta clases PHP y hojas de estilo.

2.2 - Creación

Para crear nuestra aplicación hemos tenido pues que crear un bundle, para ello utilizaremos la terminal para ir a la carpeta del proyecto, es decir, htdocs/entryLog. Acto seguido lo único que tenemos que hacer es ejecutar el siguiente comando:

```
php app/console generate:bundle
```

Una vez hecho esto tendremos que introducir los siguientes parámetros:

1. *Bundle namespace*: Este es el nombre de la carpeta que contendrá los archivos del bundle. Lo recomendado es poner el nombre del proyecto seguido del nombre del bundle y, finalmente de la palabra "Bundle". En nuestro caso quedaría de la siguiente manera: entryLog/entryLogBundle.
2. *Bundle name*: Aquí nos pide un nombre corto para el bundle, dándonos como sugerencia el nombre puesto anteriormente seguido de "Bundle", en otras palabras, entryLogBundle. Lo más adecuado para evitar posibles confusiones en el desarrollo de la aplicación es que aceptemos esa sugerencia.
3. *Target directory*: En este punto también nos hace una sugerencia sobre dónde queremos que esté el bundle. Lo más acertado es aceptar la sugerencia.
4. *Configuration format*: Lo que nos pide aquí es que definamos el formato en que estarán los archivos de configuración. Podremos elegir entre formatos como XML o YML, aunque como es natural tratándose de este trabajo indicaremos el formato PHP.
5. *Generate the whole directory structure*: Aquí podremos decidir si generar todas las carpetas de un bundle, es decir, generar incluso las carpetas precindibles para el funcionamiento del bundle. En nuestro caso respondemos afirmativamente, con el fin de conocer lo mejor posible las posibilidades de nuestro bundle.
6. *Do you confirm the generation?*: Esta es únicamente la confirmación para crearlo.
7. *Confirm automatic update of yout kernel?*: Nos permite activarlo en la aplicación automáticamente. Le ponemos que sí.
8. *Confirm automatic update of the routing?*: Nos da la oportunidad de que el enrutamiento se cargue automáticamente, a lo que también respondemos de forma afirmativa.

Tras confirmar la creación se generará la siguiente esctructura dentro de la carpeta EntryLog/src:

```
entryLogBundle
    ├── Controller
    │   ├── DefaultController.php
    │   └── entryLogController.php
    ├── DependencyInjection
    │   ├── Configuration.php
    │   └── entryLogExtension.php
    ├── entryLogBundle.php
    ├── Resources
    │   ├── config
    │   │   ├── routing.php
    │   │   └── services.php
    │   ├── doc
    │   │   └── index.rst
    │   ├── public
    │   │   ├── css
    │   │   ├── images
    │   │   └── js
    │   ├── translations
    │   │   └── messages.fr.xlf
    │   └── views
    │       ├── Default
    │       │   └── index.html.twig
    │       └── entryLog
    │           └── home.html.twig
    └── Tests
        └── Controller
            └── DefaultControllerTest.php
```

2.3 - Activación

Una vez creado lo único que deberemos hacer para que funcione es activarlo, sin embargo, al haber creado el *bundle* mediante la instrucción *generate:bundle* se ha activado automáticamente. De todas formas, si queremos activar un *bundle* externo, es decir, que no hayamos generado, sino creado de otro modo deberemos modificar el archivo AppKernel.php de la carpeta EntryLog/app.

En este fichero encontraremos una función llamada *registerBundles*, dentro del cual hay un *array* de *bundles* a la que tendremos que añadir el *bundle* en cuestión. La declaración del *array* quedaría de la siguiente manera:

```
<?php

use Symfony\Component\HttpKernel\Kernel;
use Symfony\Component\Config\Loader\LoaderInterface;

class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
            new Symfony\Bundle\SecurityBundle\SecurityBundle(),
            new Symfony\Bundle\TwigBundle\TwigBundle(),
            new Symfony\Bundle\MonologBundle\MonologBundle(),
            new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
            new Symfony\Bundle\AsseticBundle\AsseticBundle(),
            new Doctrine\Bundle\DoctrineBundle\DoctrineBundle(),
            new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
            new AppBundle\AppBundle(),
            new entryLog\entryLogBundle\entryLogBundle(),
        );
        …
        }
…
}
```

Finalmente podemos probar que ha sido correctamente creado accediendo a la ruta localhost:8000/hello/*nombre*. Esto nos llevará a una página que nos devolverá lo siguiente:

Este será una especie de HolaMundo que podremos usar como ejemplo para desarrollar nuestra propia aplicación.

3 - Enrutamiento

3.1 - Introducción

El enrutamiento nos permitirá ejecutar una función específica al acceder a una determinada ruta. De esta manera podremos crear URLs amigables y así evitar URLs complejas que contengan datos que no queramos mostrar.

Con Symfony tendremos la posibilidad de crear rutas de forma sencilla, tanto asociadas al proyecto como a un *bundle* concreto. Si lo asociamos a un *bundle* podremos tener activada la ruta únicamente si está activado, muy útil si queremos modificar un *bundle* y no queremos que la ruta funcione mientras dura el mantenimiento.

3.2 - Creación

Para introducir una ruta tendremos que modificar el archivo *routing*, situado en la carpeta EntryLog/Resources/config, añadiendo lo siguiente:

PHP – Asociado al *bundle*

```
$collection->add('nombre_ruta', new Route('ruta', array(
    '_controller' => 'EntryLogBundle:Controlador:Action,
)));
```

YML – Asociado al proyecto, pues hemos creado el *bundle* con archivos PHP y el proyecto es creado por defecto en YML

```
nombre_ruta:
    path:      ruta
    defaults:  { _controller: EntryLogBundle: Controlador:Action }
```

En nuestra aplicación crearemos las siguientes rutas en el archivo PHP, una por cada página de la aplicación:

```
$collection->add('tfg_home', new Route('/MiTfg', array(
    '_controller' => 'TfgBundle:Default:home',
)));

$collection->add('tfg_busqueda', new Route('/busqueda', array(
    '_controller' => 'TfgBundle:Default:busqueda',
)));

$collection->add('tfg_socios', new Route('/socios', array(
    '_controller' => 'TfgBundle:Default:socios',
)));

$collection->add('tfg_historial', new Route('/bhistorial', array(
    '_controller' => 'TfgBundle:Default:borrar_historial',
)));

$collection->add('tfg_crearsocio', new Route('/crearSocio', array(
    '_controller' => 'TfgBundle:Default:crear_socio',
)));
```

3.3 - Variables

Mientras introducimos las rutas veremos que ya hay una ruta creada, a la que hemos accedido antes para comprobar que el *bundle* estaba activado:

```
$collection->add('tfg_hello', new Route('/hello/{name}', array(
    '_controller' => 'TfgBundle:Default:index',
)));
```

Hay una diferencia en esta ruta y es la parte dinámica, una variable por así decirlo. Esta se encuentra al final de la ruta y está delimitada por dos llaves *{name}*. La variable de la ruta será recibida por la función *action* pudiendo mostrar diferentes datos dependiendo de la variable mandada.

Aunque en un principio puede parecer que esto va en contra del objetivo del enrutamiento, puesto que la variable aparece en la ruta, podemos redirigir a otra ruta distinta desde el controlador. Sin embargo sí que será visible cuando el usuario actué sobre el elemento que lleva a la ruta, pero es un precio pequeño comparado con la gran funcionalidad que ofrece.

La función *action* del controlador tendría una estructura similar a esta:

```
public function indexAction($name)
{
    return $this->render('entryLogBundle:Default:index.html.twig', array('name' => $name));
}
```

Como vemos en este caso la función recoge la variable mandada y la manda, a su vez, al archivo TWIG, es decir, al *template*, que contiene lo siguiente:

```
Hello {{ name }}!
```

En la aplicación desarrollada usaremos esta valiosa herramienta para las acciones de modificar y eliminar un socio. Nos bastará una sola ruta para cada acción y usaremos como variable el identificador único de cada socio:

```
$collection->add('tfg_modificarsocio', new Route('/modificarSocio/{id}', array(
    '_controller' => 'TfgBundle:Default:modificar_socio',
)));

$collection->add('tfg_borrarsocio', new Route('/borrarSocio/{id}', array(
    '_controller' => 'TfgBundle:Default:borrar_socio',
)));
```

4 - Entidades

4.1 - Introducción

Entidad es el término que utiliza Symfony para denominar los modelos, es decir, fijan una estructura para la información proveniente de la base de datos para poder utilizarla de una manera más eficaz en el código. A efectos prácticos y, hablando desde el punto de vista del paradigma de orientación a objetos, una entidad no es más que una clase.

4.2 - Creación

Una de las mejores funcionalidades que tiene Symfony desde el punto de vista, tanto de bases de datos como de la orientación a objetos es que podemos crear una clase y enlazarle una tabla en la base de datos con un solo comando de consola.

También podremos hacer esto manualmente, pero vemos mucho más práctico y didáctico usar la herramienta que Symfony nos proporciona para dicho fin. Esta herramienta es Doctrine y como hemos visto anteriormente es una librería que nos servirá de middleware entre la base de datos y el código.

Para crear una entidad iremos a la carpeta del proyecto EntryLog y ejecutaremos el siguiente comando:

```
php app/console doctrine:generate:entity
```

A continuación, al igual que en el caso de la creación de *bundles*, nos pedirá que introduzcamos varios parámetros:

1. *Entity shortcut name*: Este será el nombre corto de la entidad, el cual tendremos que introducir concatenado con el nombre del *bundle* al que pertenece, en nuestro caso quedaría así: TfgBundle:Registro.
2. *Configuration format (yml, xml or annotation)*: Aquí podremos definir el formato en que queremos mapear la información de la base de datos. En nuestro caso usaremos anotaciones, la opción por defecto.

Lo último que tendremos que especificar son los atributos que queremos que tenga nuestra entidad Registro. Tenemos que tener en cuenta que al crear la entidad tendremos creado un atributo por defecto que será el id, la que nos permitirá identificar cada uno de los objetos creados.

Para el resto de atributos, primero especificaremos el nombre, teniendo cuidado de no introducir una palabra reservada de la base de datos, pues el nombre introducido será el usado en la base de datos y puede provocar fallos en la futura creación de objetos.

Una vez fijado el nombre determinaremos el tipo de atributo, introduciendo una de las posibilidades que nos lista Symfony, aunque si queremos que sea de un tipo no listado, como otra entidad, deberemos modificarla desde el código.

Al definir el tipo es posible que nos pregunte algo más, dependiendo del tipo que hayamos elegido, por ejemplo, si elegimos *string*, nos preguntará el tamaño máximo de este.

En cuanto hayamos definido todos los atributos deseados basta con introducir uno nuevo con el nombre vacío, tras lo cual nos preguntará si queremos generar un repositorio vacío, a lo que responderemos que no, ya que no nos hará falta. Por último nos hará la típica pregunta de confirmación para crear la entidad, naturalmente respondemos que sí.

```
This command helps you generate Doctrine2 entities.

First, you need to give the entity name you want to generate.
You must use the shortcut notation like AcmeBlogBundle:Post.

The Entity shortcut name: entryLogBundle:Registro

Determine the format to use for the mapping information.

Configuration format (yml, xml, php, or annotation) [annotation]:

Instead of starting with a blank entity, you can add some fields now.
Note that the primary key will be added automatically (named id).

Available types: array, simple_array, json_array, object,
boolean, integer, smallint, bigint, string, text, datetime, datetimetz,
date, time, decimal, float, blob, guid.

New field name (press <return> to stop adding fields): usuario
Field type [string]:
Field length [255]:

New field name (press <return> to stop adding fields): idUsuario
Field type [string]: integer

New field name (press <return> to stop adding fields): fecha
Field type [string]: datetime

New field name (press <return> to stop adding fields): accion
Field type [string]:
Field length [255]:

New field name (press <return> to stop adding fields): url
Field type [string]:
Field length [255]:

New field name (press <return> to stop adding fields):

Do you want to generate an empty repository class [no]?

  Summary before generation

You are going to generate a "entryLogBundle:Registro" Doctrine2 entity
using the "annotation" format.

Do you confirm generation [yes]?

  Entity generation
```

Ahora ya tenemos creada la entidad, podemos comprobarlo viendo que se ha creado la carpeta Entity dentro del *bundle* (en caso de que no existiera) y que dentro está la clase creada Registro.php:

```
<?php

namespace TfgBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Registro
 *
 * @ORM\Table()
 * @ORM\Entity
 */
class Registro
{
        /**
        * @var integer
        *
        * @ORM\Column(name="id", type="integer")
        * @ORM\Id
        * @ORM\GeneratedValue(strategy="AUTO")
        */
        private $id;

         /**
        * @var string
        *
        * @ORM\Column(name="usuario", type="string")
        */
        private $usuario;

        /**
         * @var integer
        *
        * @ORM\Column(name="idUsuario", type="integer")
         */
        private $idUsuario;

        /**
        * @var \DateTime
        *
        * @ORM\Column(name="fecha", type="datetime")
        */
        private $fecha;

        /**
        * @var string
        *
        * @ORM\Column(name="accion", type="string", length=255)
        */
        private $accion;

        /**
        * @var string
        *
        * @ORM\Column(name="url", type="string", length=255)
        */
        private $url;
        …
}
```

Dentro de esta también tendremos creados los métodos consultores y modificadores, son métodos muy sencillos, si se necesita algún comportamiento más complejo tendremos que hacerlo manualmente:

```
/**
 * Get id
 *
 * @return integer
 */
public function getId()
{
    return $this->id;
}

/**
 * Set usuario
 *
 * @param string $usuario
 * @return Registro
 */
public function setUsuario($usuario)
{
    $this->usuario = $usuario;

    return $this;
}

/**
 * Get usuario
 *
 * @return string
 */
public function getUsuario()
{
    return $this->usuario;
}

…
```

Podemos observar que de los atributos creados hay métodos consultores y modificadores, mientras que del identificador solo hay método consultor. La gestión de este atributo estará controlada completamente por Symfony, evitando así errores de programación que pueden provocar la pérdida del objeto.

4.3 - Configuración de la entidad en la base de datos

Aunque ya hayamos creado la entidad, no ha sido creada la tabla correspondiente en la base de datos, así pues, deberemos introducir el siguiente comando:

```
php app/console doctrine:schema:update --force
```

De este modo Doctrine revisa la información de mapeo de todas las entidades creadas para así actualizar la base de datos creando o modificando las tablas correspondientes. Tras ejecutar este comando, dispondremos de una nueva tabla en la base de datos, completamente funcional, y cuyas columnas coinciden con los metadatos especificados en la clase Registro.

Si por alguna razón debemos añadir o eliminar atributos de una clase o modificar su tipo será necesario que ejecutemos de nuevo este comando, de no hacerlo se producirán graves errores.

4.4 – Persistencia de objetos

Como ya hemos dicho, la gran ventaja de la gestión de entidades que nos proporciona Symfony es que podemos introducir información en la base de datos sin necesidad de realizar una sola instrucción SQL. La librería Doctrine nos ofrece las herramientas necesarias para que incluso alguien sin conocimientos de bases de datos pueda persistir un objeto. A continuación mostraremos cómo, primero creamos el objeto y le damos valor a sus atributos:

```
public function createAction()
{
        $entry = new Registro();
        $entry->setIp('127.0.0.1');
        …
```

Lo que estamos haciendo es asociar la creación de un objeto a la función *action*, al tratarse de un controlador podremos obtener la clase Doctrine, y el gestor de entidades de este, para utilizarlo:

```
$entityManager = $this->getDoctrine()->getManager();
```

El gestor de entidades de Doctrine nos servirá tanto para introducir como para obtener información de la base de datos. Ahora que ya lo tenemos podremos guardar el objeto con la función *persist()*:

```
$entityManager->persist($entry);
$entityManager->flush();
```

Sin embargo, sólo con la función *persist()* no basta, pues esta no genera ninguna sentencia SQL. Para que se produzca la persistencia tendremos que usar también la función *flush()*. Cuando se llama a esta función, Doctrine examina todos los objetos gestionados para ver si debe persistirlos. Además gracias a esta función Doctrine calcula cual es la manera más eficaz de generar todas las sentencias SQL necesarias, si en vez de introducir un objeto se van a introducir varios, Doctrine preparará un sola consulta y la reutilizará las veces que hagan falta. Este comportamiento corresponde con el patrón de diseño *Unit of Work*.

Otra característica interesante de esta función es que, dependiendo del objeto a introducir, identificará si se trata de una sentencia INSERT o de un UPDATE, es decir, si se trata de un objeto nuevo o de uno ya existente.

Una vez visto cómo crear y modificar un registro procederemos a explicar cómo consultarlos y borrarlos. Para realizar consultas primero tendremos que obtener un repositorio sobre el que hacer consultas.

En el caso de Doctrine, al hacer consultas las haremos hacia un determinado tipo de objetos, es decir, el repositorio obtenido solo podrá contener información de una determinada clase, es decir, para realizar una consulta tendríamos que escribir el siguiente código:

```
$repository = $this->getDoctrine()->getRepository('TfgBundle:Busqueda');
$historial = $repository->findAll();
```

Tal como se muestra, para obtener el repositorio tenemos que especificar la clase sobre la que haremos consultas. También podemos ver la función usada, *findAll()* con la que obtendremos todos los registros de una clase, aunque podremos realizar consultas más específicas como buscar por un determinado atributo *findByAtributo(filtro)*, o por el identificador único, *find($id)*.

Otra cosa a tener en cuenta es que la consulta es la única instrucción en base de datos en la que no hace falta usar la función *flush()*, puesto que la base de datos no se modifica durante las consultas.

Por último, para eliminar un registro usaremos la función *remove()*, del mismo modo que antes usábamos *persist()*. En este ejemplo, primero obtendremos el objeto y luego lo eliminaremos:

```
$em = $this->getDoctrine()->getManager();
$repository = $this->getDoctrine()->getRepository('TfgBundle:Socio');
$socio = $repository->find($id);

$em->remove($socio);
$em->flush();
```

5 - Vistas

5.1 - Introducción

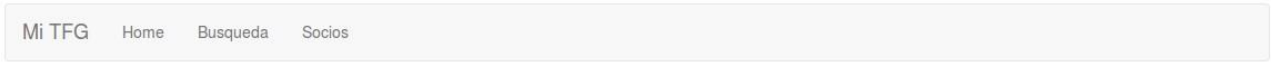
Las vistas son los ficheros que devolverá el controlador y que ofrecerá al usuario una interfaz sobre la que actuar. Estas vistas en Symfony consistirán en plantillas TWIG, las cuales nos dan la oportunidad de utilizar la programación HTML junto a algunas funciones de control.

5.2 - Estructura

Estos archivos están estructurados por bloques, fragmentos de código con un determinado fin que podrán ser reutilizados en plantillas que hereden de esta. Esto quiere decir que podremos usar una plantilla como base para nuestra aplicación y hacer que el resto hereden de ella, muy útil para definir las hojas de estilo y los archivos JavaScript una sola vez o para crear partes fijas de las páginas.

Esta plantilla base estará asociada al proyecto, no al *bundle*, debido a que lo normal es que un proyecto tenga varios *bundles*, pero un determinado aspecto para todos. Así pues la plantilla base estará situada en la carpeta app/Resources/views.

5.2.1 - Vista Base



Aquí mostramos el código del fichero, al cual hemos denominado tfgBase.html.twig:

```
<html>
   <head>
      <meta charset="UTF-8"/>
      <meta name="viewport" content="width=device-width, initial-scale=1">
      <title>
         {% block title %}Tfg Home{% endblock %}
      </title>
      {% block stylesheets %}
         <link href="{{ asset('bundles/tfg/css/bootstrap/bootstrap.css') }}" type="text/css" rel="stylesheet" />
         <link href="{{ asset('bundles/tfg/css/tfg.css') }}" type="text/css" rel="stylesheet" />
      {% endblock %}
   </head>
   <body>
      <div class="container">
      {% block menu %}
         <nav class="navbar navbar-default">
            <div class="container-fluid">
               <div class="navbar-header">
                  <a class="navbar-brand" href="/MiTfg">
                     Mi TFG
                  </a>
               </div>
               <div id="navbar" class="navbar-collapse collapse">
                  <ul class="nav navbar-nav">
                     <li>
                        <a href="/MiTfg">
                           Home
                        </a>
                     </li>
                     <li>
                        <a href="/busqueda">
                           Busqueda
                        </a>
                     </li>
                     <li>
                        <a href="/socios">
                           Socios
                        </a>
                     </li>
                  </ul>
               </div>
            </div>
         </nav>
      {% endblock %}
      {% block body %}{% endblock %}
      </div>
   </body>
</html>
```

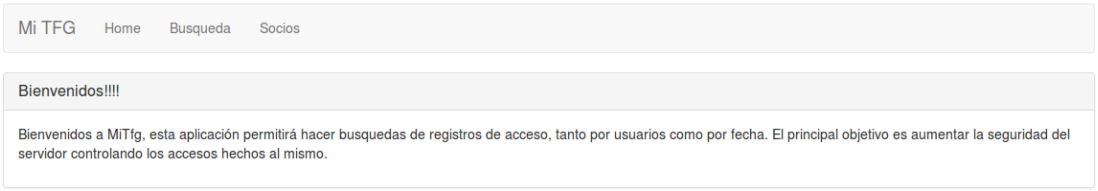
Se puede observar que en esta plantilla hemos definido todo excepto el bloque *body*, esto es debido a que esta plantilla nunca se mostrará sola, siempre será mostrada mediante otra plantilla, que heredará de esta y que tendrá su propio contenido. Por otra parte, también podremos sobrescribir bloques heredados, basta con escribir un bloque que tenga el mismo nombre y rellenarlo con lo que queramos.

Para hacer que una plantilla herede de otra es necesario escribir este código:

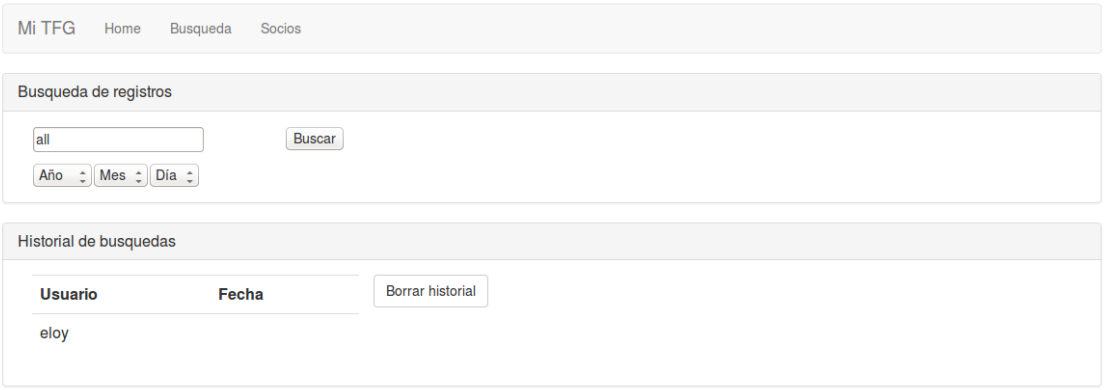
```
{% extends '::tfgBase.html.twig' %}
```

Y de esta manera creamos las siguientes plantillas a partir de esta:

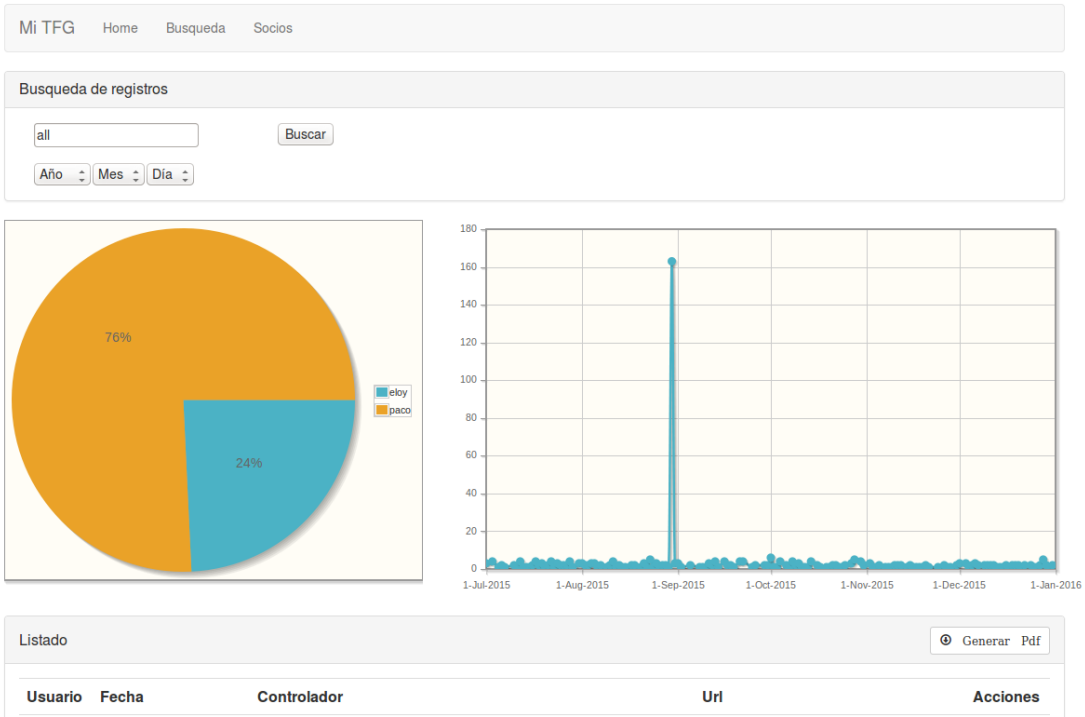
5.2.2 - Vista Bienvenida



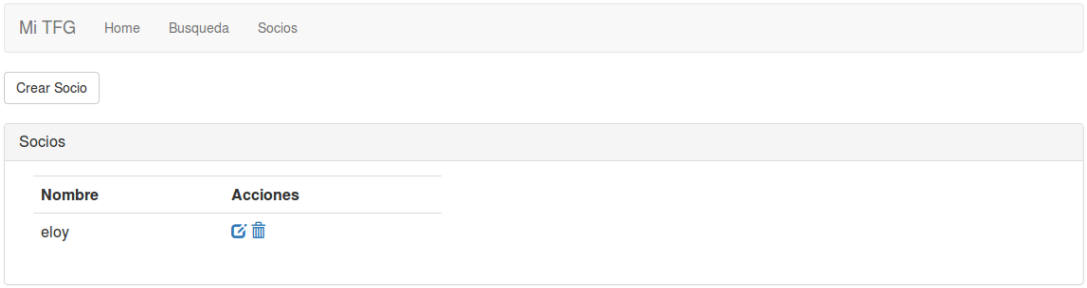
5.2.3 - Vista Búsqueda



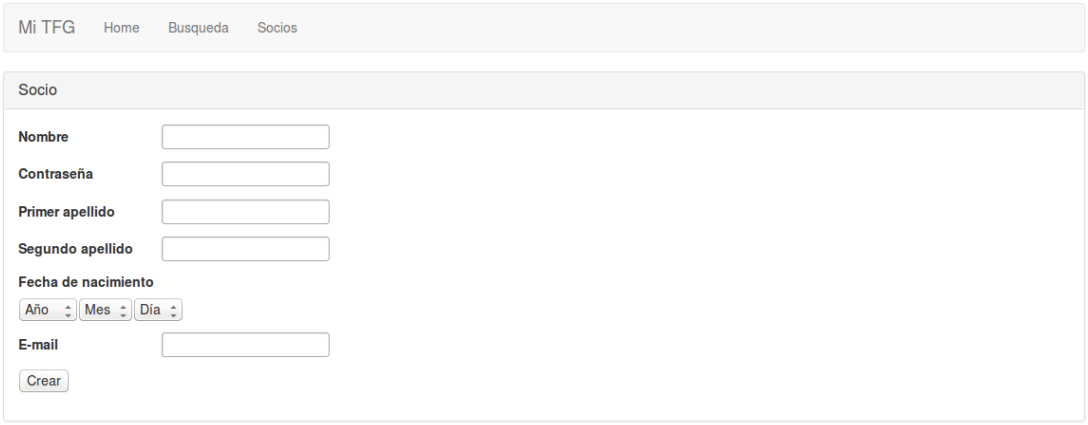
5.2.4 – Vista Listado



5.2.5 – Vista Socios



5.2.6 - Crear/ Modificar Socio



En el caso de la página de bienvenida la información contenida será muy básica, únicamente una presentación no dinámica y que no actuará sobre la base de datos. Sin embargo las demás páginas sí que se servirán de información generados por el usuario y para ello tendremos que hacer uso de variables PHP.

5.3 – Variables e instrucciones de flujo

Para usar variables PHP en las plantillas tendremos que recurrir al controlador. Este no solo se encarga de devolver la plantilla ante una petición, sino que también podrá devolverla enviándole variables.

Así pues, una función *action* de nuestro controlador tendrá un aspecto como este:

```
public function sociosAction()
    {
        $repository = $this->getDoctrine()->getRepository('TfgBundle:Socio');
        $socios = $repository->findAll();

        return $this->render('TfgBundle:TfgViews:tfgSocios.html.twig', array(
            'socios' => $socios
        ));
    }
```

Como se puede ver, esta función devuelve la plantilla “tfgSocios.html.twig”, a la que envía un listado de socios para que esta los muestre. A continuación mostraremos el código con el que el fichero TWIG realiza su labor:

```
<table class="table">
	<th>
		Nombre
	</th>
	<th>
		Acciones
	</th>
	{% for socio in socios %}
		<tr>
			<td>
				{{ socio.nombre }}
			</td>
			<td>
				<a href="/modificarSocio/{{ socio.id }}" class="glyphicon glyphicon-edit"></a>
				<a href="borrarSocio/{{ socio.id }}" class="glyphicon glyphicon-trash"></a>
			</td>
		</tr>
	{% endfor %}
</table>
```

Para mostrar todos los elementos del listado se utilizará la instrucción *for*, que tendrá un comportamiento similar a un *foreach*. Las funciones irán encapsuladas entre llaves y porcentajes y las variables entre dobles llaves y pudiendo tratarse tanto de variables recibidas como atributos pertenecientes a objetos.

Además de esta hay unas pocas funciones más que podremos utilizar en las plantillas, aunque en esta aplicación las únicas utilizadas han sido las ya nombradas. No obstante mencionaremos un par de ellas brevemente.

Como es natural también contaremos con la instrucción *if* con la que podremos comparar las variables enviadas por el controlador, pero también podremos utilizarla para actuar sobre variables internas del propio fichero, variables que podemos crear mediante la instrucción *set*.

```
{% set contador  = 1 %}
{% if var > 1 %}
	…
	…
{% endif %}
```

Con la instrucción *set* también podremos capturar fragmentos completos de código como se muestra a continuación:

```
{% set contenedor %}
	<div>
		…
		…
	</div>
 {% endset %}
```

6 – Registros

6.1 - Introducción

Los servidores web suele guardar un registro de accesos entre sus archivos, un listado de entradas donde se suelte especificar la ip que accedió, el momento del acceso y la *url* accedida.

Los proyectos Symfony tienen su propio registro de acceso y proporciona algún dato adicional propio del proyecto, como el nombre interno de la ruta accedida o el controlador accionado. Al igual que en los servidores podemos configurar dicho registro de forma que nos dé mucha más información y es precisamente esta capacidad la que nos hará conseguir los datos necesarios para nuestra aplicación.

6.2 – Formato

En primer lugar, el registro se encuentra en la carpeta app/logs de nuestro proyecto y recibe el nombre de app_dev.log. Consiste en un *array* de cadenas, cada una perteneciente a una notificación. Las notificaciones sobre las que nos centraremos serán aquellas encabezadas por "request.INFO: Matched route", las cuales serán imprimidas cuando el socio acceda a una ruta existente. Aquí podemos ver un ejemplo de este tipo de entradas:

```
[2015-08-29 10:06:56] request.INFO: Matched route "_configurator_home".
{"route_parameters":{"_controller":"Sensio\\Bundle\\DistributionBundle\\Controller\\
ConfiguratorController::checkAction","_route":"_configurator_home"},
"request_uri":"http://localhost/prueba/web/app_dev.php/_configurator/"} []
```

En el registro las entradas estarán en una sola línea, pero la hemos organizado en distintas líneas para poder diferenciar mejor los distintos tipos de información que nos proporciona. En la primera línea vemos la fecha y la hora del servidor en el momento del acceso, así como el nombre de la ruta accedida.

En el resto de líneas vemos los parámetros de la ruta, como el controlador que gestiona la petición y la acción que se ejecuta y, de nuevo, el nombre de la ruta. El último parámetro que nos muestra es la *url* accedida, es decir, la ruta completa que ve el socio cuando accede.

Pese a todo, aquí no tenemos la suficiente información para saber qué socio ha accedido a dicha ruta, por tanto, tendremos que configurar la función que inserta esta información en el registro, aunque en primer lugar veremos cómo obtener la información que queremos introducir. La manera más común de diferenciar entre los actos de los usuarios de un sitio web es mediante el uso de sesiones.

De esta forma, cuando un usuario inicie sesión estas serán las acciones a realizar:

```
public function iniciarSesionAction($name, $id, Request $request)
{
        $session->clear();
        $session->set('tfg_idUser', $id);
        $session->set('tfgUser', $name);
        …
 }
```

Nos encontramos en un punto donde ya hemos comprobado la identidad del usuario mediante la contraseña y escribimos sus datos en la sesión, para poder utilizarlos mientras esta esté activa. Como es normal podremos usar otros nombres de variables o guardar otro tipo de información, pero nosotros usaremos esta en concreto para el desarrollo de la aplicación

Por último, hay que tener en cuenta que esta acción no ha sido descrita en los casos de uso ni en ningún otro lado debido a que se ha implementado para hacer pruebas de funcionamiento únicamente.

Una vez guardada la información en la sesión la imprimiremos junto a la entrada anterior para poder obtenerla desde la aplicación. Para ello modificaremos el archivo que escribe la entrada de la ruta en el registro, RouterListener.php, ubicado en el directorio "vendor/symfony/symfony/src/Symfony/Component/HttpKernel/EventListener/". Dentro de esta clase hay una función llamada *onKernelRequest(Request $request)*, la cual modificaremos de la siguiente manera:

```
if (null !== $this->logger) {
$this->logger->info(sprintf('Matched route "%s".', isset($parameters['_route']) ?
$parameters['_route'] : 'n/a'), array(
                'route_parameters' => $parameters,
                'request_uri' => $request->getUri(),
                'request_session' => $request->getSession(),
            ));
}
```

La modificación consiste en la introducción del último elemento del *array*, la sesión que obtenemos del *request*, cuya información será imprimida en el registro con la cabecera 'request_session' de la siguiente manera:

```
[2015-08-30 18:30:50] request.INFO: Matched route "_wdt".
{"route_parameters":{"_controller":"web_profiler.controller.profiler:toolbarAction","token":"d1d941","_route":"
_wdt"},
"request_uri":"http://localhost:8000/_wdt/d1d941",
"request_session":{"tfg_idUser":1,"tfgUser":"eloy"}} []
```

Hecho esto, ya podremos leer del fichero la información que necesitamos.

6.3 – Lectura y muestra

Para leer el registro de accesos y obtener información de él usaremos la clase búsqueda, a la cual le hemos añadido los métodos necesarios para buscar dentro de cada registro la información pertinente e introducirla en un objeto de tipo Registro, para así gestionarla mejor.

El primer paso es llamar a la función de la clase Búsqueda desde el controlador:

```
if($form->isValid())
{
        $em = $this->getDoctrine()->getManager();
        $em->persist($busqueda);
        $em->flush();

        $entries = $busqueda->getEntries();
        …
```

Esta clase devolverá un *array* de objetos tipo Registro que después enviaremos a la plantilla del listado, pero antes veremos cómo se rellena dicho *array*. Una vez llamada, la función *getEntries()* se subdivide en cuatro funciones de la siguiente manera:

```
$all = $this->user=='all';
$always = !isset($this->date);

if(!$all && !$always)
{
        return $this->getEntriesByUserAndDate();
}
if($all && $always)
{
        return $this->getAllEntries();
}
if(!$all)
{
        return $this->getEntriesByUser();
}
if(!$always)
{
        return $this->getEntriesByDate();
 }
return null;
```

Esto es debido a que hay cuatro tipos de búsqueda, filtrando por socio, filtrando por fecha, filtrando por ambos o sin filtros. Aunque por nuestra parte requiere el uso de una mayor cantidad de código, ofrecerá más rapidez a la hora de ejecutarse.

Usaremos la función de búsqueda con ambos filtros para mostrar cómo filtramos las entradas:

```
public function getEntriesByUserAndDate()
    {
        $file = '../app/logs/dev.log';
        if(file_exists($file))
        {
            $log = file($file);

            $entries = array();

            $user = $this->user;
            $date = (string)date_format($this->date,"Y-m-d");

            foreach ($log as $entry)
            {
                if(strpos ($entry, 'tfg_idUser') && strpos($entry, $date) && strpos($entry, '"tfgUser":"'.$user.'"'))
                {
                    $accion = $this->getEntryControlador($entry);
                    $url = $this->getEntryUrl($entry);
                    $entryDate = $this->getEntryDate($entry);
                    $id_user = $this->getEntryId($entry);


                    $registro = new Registro();
                    $registro->setIdUsuario($id_user);
                    $registro->setAccion($accion);
                    $registro->setFecha($entryDate);
                    $registro->setUrl($url);
                    $registro->setUsuario($user);

                    $entries[]=$registro;
                }
            }
        }
        return $entries;
    }
```

Tal como se puede observar, lo primero que haremos es obtener el fichero de registros, usando, como medida de seguridad contra errores, una comprobación previa de la existencia de dicho fichero.

Seguidamente crearemos un *array* vacío y obtendremos los términos introducidos por el usuario, para lo cual la creación de la clase búsqueda es muy útil, pues solo tendremos que obtener el atributo del mismo objeto que estamos manejando ($this->atributo).

Al tratarse de un archivo log lo trataremos como un *array* de entradas que hay que recorrer. Por cada entrada aplicaremos los filtros que procedan, pero antes de eso comprobaremos que se trata de una entrada válida para nosotros *strpos($entry, 'tfg_idUser')*. Para las otras comparaciones usaremos la misma función, la cual nos devolverá la posición de la cadena introducida y, en caso de no encontrarla, devolverá 0 o lo que es lo mismo, *false*.

Una vez que sabemos que es una entrada válida procederemos a “desmembrarla” para obtener los distintos datos que luego mostraremos al usuario:

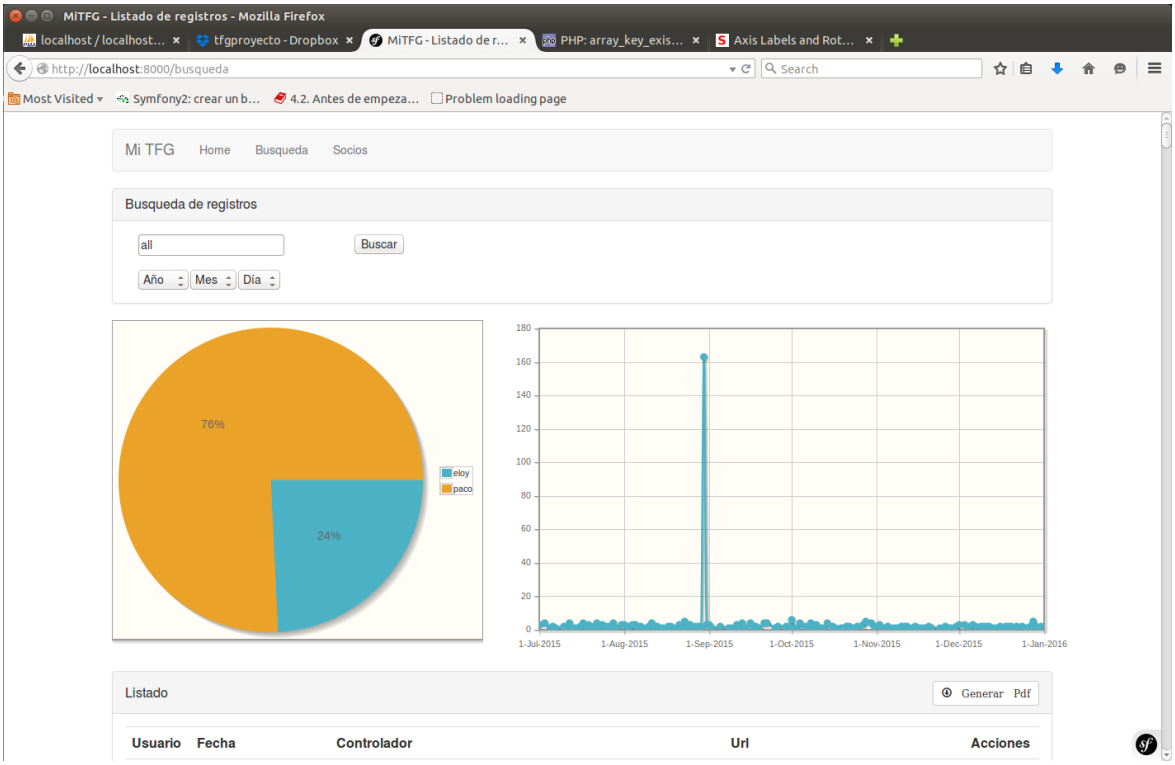
```
public function getEntryControlador($entry)
    {
        $ini = (int)strpos($entry, '_controller":')+14;
        $end = (int)strpos($entry, 'Action');
        $length = $end - $ini + 6;

        $controlador = substr($entry, $ini, $length);
        return $controlador;
    }

    public function getEntryUrl($entry)
    {
        $ini = (int)strpos($entry, 'request_uri":"')+14;
        $end = (int)strpos($entry, '","request_session');
        $length = $end - $ini;

        $url = substr($entry, $ini, $length);
        return $url;
    }

    public function getEntryUser($entry)
    {
        $ini = (int)strpos($entry, 'tfgUser') + 10;
        $end = (int)strpos($entry, '"', $ini);
        $length = $end - $ini;

        $user = substr($entry, $ini, $length);

        return $user;
    }

    public function getEntryId($entry)
    {
        $ini = (int)strpos($entry, 'tfg_idUser') + 12;
        $end = (int)strpos($entry, ',"tfgUser');
        $length = $end - $ini;

        $id = (int)substr($entry, $ini, $length);

        return $id;
    }

    public function getEntryDate($entry)
    {
        $date = substr($entry, 1, 10);
        $time = substr($entry, 12, 8);
        $datetime = new \DateTime($date.' '.$time);

        return $datetime;
    }
```

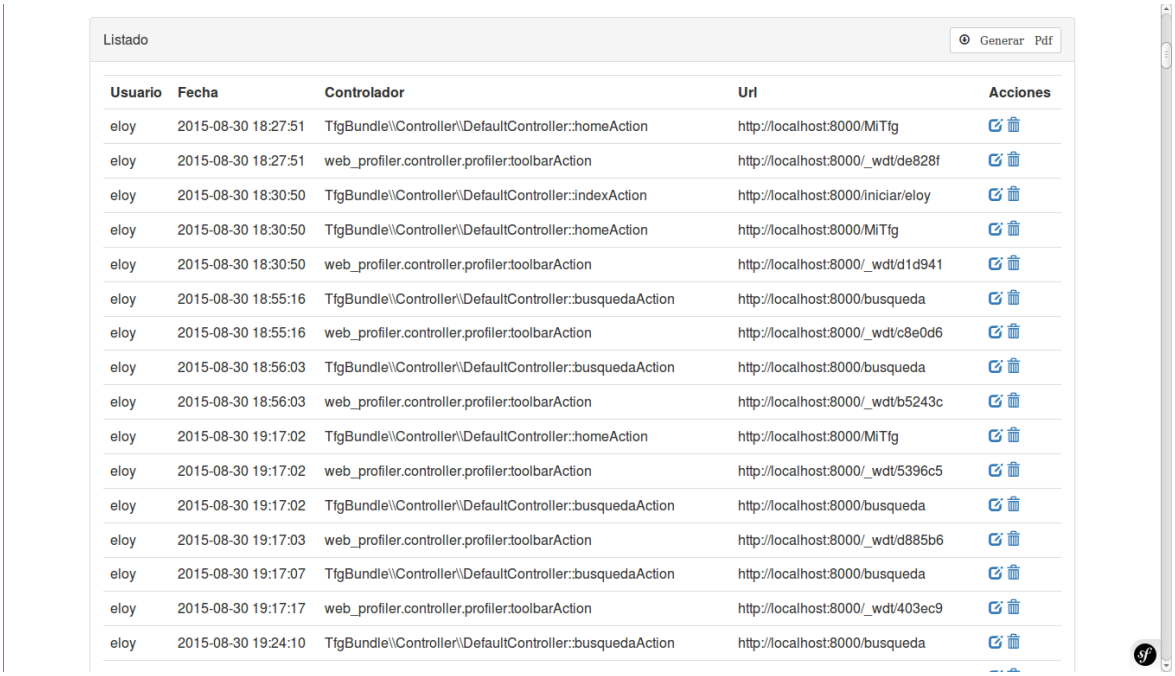
Estas funciones consistirán básicamente en buscar la aparición del nombre del atributo y a partir de esta obtener el valor asociado. Pese a tener la fecha de la búsqueda la buscaremos de nuevo en la entrada del registro para obtener también la hora y crear un objeto DateTime más completo que el que guarda la búsqueda.

En cuanto tengamos todos los datos crearemos un objeto de tipo Registro y lo rellenaremos mediante los métodos modificadores creados automáticamente al crear la entidad. Este objeto

no será persistido, en otras palabras, no será guardado en la base de datos, su función es simplemente ofrecer una estructura definida para cada entrada del registro.

Finalmente, al haber obtenido todos los registros, el *array* será devuelto a la función *getEntries()* y será redirigido al controlador. Este lo enviará a la plantilla junto a listados modificados para ser leídos por las gráficas mostradas, dando como resultado una página con el siguiente aspecto:





Listado — Generar Pdf

Usuario	Fecha	Controlador	Url	Acciones
eloy	2015-08-30 18:27:51	TfgBundle\\Controller\\DefaultController::homeAction	http://localhost:8000/MiTfg	
eloy	2015-08-30 18:27:51	web_profiler.controller.profiler:toolbarAction	http://localhost:8000/_wdt/de828f	
eloy	2015-08-30 18:30:50	TfgBundle\\Controller\\DefaultController::indexAction	http://localhost:8000/iniciar/eloy	
eloy	2015-08-30 18:30:50	TfgBundle\\Controller\\DefaultController::homeAction	http://localhost:8000/MiTfg	
eloy	2015-08-30 18:30:50	web_profiler.controller.profiler:toolbarAction	http://localhost:8000/_wdt/d1d941	
eloy	2015-08-30 18:55:16	TfgBundle\\Controller\\DefaultController::busquedaAction	http://localhost:8000/busqueda	
eloy	2015-08-30 18:55:16	web_profiler.controller.profiler:toolbarAction	http://localhost:8000/_wdt/c8e0d6	
eloy	2015-08-30 18:56:03	TfgBundle\\Controller\\DefaultController::busquedaAction	http://localhost:8000/busqueda	
eloy	2015-08-30 18:56:03	web_profiler.controller.profiler:toolbarAction	http://localhost:8000/_wdt/b5243c	
eloy	2015-08-30 19:17:02	TfgBundle\\Controller\\DefaultController::homeAction	http://localhost:8000/MiTfg	
eloy	2015-08-30 19:17:02	web_profiler.controller.profiler:toolbarAction	http://localhost:8000/_wdt/5396c5	
eloy	2015-08-30 19:17:02	TfgBundle\\Controller\\DefaultController::busquedaAction	http://localhost:8000/busqueda	
eloy	2015-08-30 19:17:03	web_profiler.controller.profiler:toolbarAction	http://localhost:8000/_wdt/d885b6	
eloy	2015-08-30 19:17:07	TfgBundle\\Controller\\DefaultController::busquedaAction	http://localhost:8000/busqueda	
eloy	2015-08-30 19:17:17	web_profiler.controller.profiler:toolbarAction	http://localhost:8000/_wdt/403ec9	
eloy	2015-08-30 19:24:10	TfgBundle\\Controller\\DefaultController::busquedaAction	http://localhost:8000/busqueda	

7 – Seguridad

7.1 – Introducción

Aparte de las herramientas de seguridad que nosotros podamos brindarle a la aplicación Symfony nos ofrece algunas posibilidades en este ámbito que nos ha parecido interesante exponer.

7.2 – CSRF

Cross-site request forgery es conocido por ser una forma en que un usuario malintencionado intenta que los demás proporcionen datos que no es recomendable que compartan. Para frustrar estos ataques Symfony nos ofrece por defecto *tokens* de seguridad CSRF.

Estos *tokens* serán enviados junto a los formularios para validar el envío de datos. Como hemos dicho antes esta "clave" se genera por defecto, sin embargo podemos configurar nosotros mismos el CSRF para elegir la clave que queramos. Esto se consigue mediante la página de configuración de Symfony, mostrada anteriormente.

7.3 – Autenticación y formularios de acceso

Aunque en esta aplicación no hemos utilizado las herramientas de validación de usuarios, mostraremos básicamente como funciona, así como las posibilidades que ofrece.

En primer lugar, en Symfony, la configuración de seguridad se guarda en el archivo security.yml, situado en la carpeta /app del proyecto. En este fichero podemos definir usuarios y roles, así como rutas configuradas para un determinado rol de una forma fácil. A continuación mostramos un ejemplo de cómo:

```
# app/config/security.yml
security:
    firewalls:
        secured_area:
            pattern:    ^/
            anonymous: ~
            http_basic:
                realm: "Secured Demo Area"

    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }
        - { path: ^/admin$, roles: ROLE_ADMIN }

    providers:
        in_memory:
            memory:
                users:
                    ryan:  { password: ryanpass, roles: 'ROLE_USER' }
                    admin: { password: kitten, roles: 'ROLE_ADMIN' }

    encoders:
        Symfony\Component\Security\Core\User\User: plaintext
```

En este ejemplo en particular vemos que se ha definido una ruta "/admin", a la cual solo podrán acceder los usuarios con el rol "ROLE_ADMIN". De esta forma, cuando alguien acceda a esta ruta se le mostrará un básico cuadro de dialogo para introducir usuario y contraseña y

estos datos serán comparados con los especificados más abajo, donde vemos que hay dos usuarios creados, uno con el rol de administrador y otro con el de usuario. El resto de rutas estarán libres de este control, es decir, no se mostrará ningún cuadro de dialogo.

Para usar un formulario más complejo podremos añadir una ruta de login y otra para la comprobación:

```
secured_area:
        pattern:    ^/
        anonymous:  ~
        form_login:
          login_path:  login
          check_path:  login_check
```

Esto nos permitirá personalizar el acceso mediante un formulario cuya plantilla podremos crear como cualquier otra. También tendremos que crear una función *action* para el formulario, pero en el solo tendremos que comprobar los errores y actuar sobre ellos. En otras palabras, si el usuario y la contraseña son correctos, Symfony se ocupa de todo, si no es así nuestra función gestionará los errores surgidos.

6. Pruebas

1 – Introducción

Para comprobar el correcto funcionamiento de la aplicación hemos tenido que realizar una serie de pruebas, así como algunas funciones que nos permitieran emular el comportamiento de la aplicación y así tener saber cómo actuaría la aplicación en diversos casos.

2 – Sesiones

Para poder leer la información de una sesión en el registro, antes teníamos que introducir en dicha sesión la información que queríamos leer, es decir, teníamos que iniciar sesión con un usuario y realizar una serie de acciones para cerciorarnos de que estas iban a estar reflejadas en el registro.

Para ello hemos creado la función *iniciarAction()*, la cual obtenía un nombre de la ruta e iniciaba sesión introduciendo como atributos el nombre y el id del socio al que perteneciera:

```
public function iniciarAction($name, Request $request)
    {
        $repository = $this->getDoctrine()->getRepository('TfgBundle:Socio');
        $socio = $repository->findByNombre($name)[0];
        $id = $socio->getId();

        $session = $request->getSession();
        $session->clear();
        $session->set('tfg_idUser', $id);
        $session->set('tfgUser', $name);

        return $this->redirectToRoute('tfg_home');
    }
```

De esta manera, al introducir la ruta */iniciar/$name*, definida en el fichero de enrutamiento, iniciábamos una sesión y navegábamos por la aplicación. Luego mirábamos en el *log* si los datos habían sido registrados.

3 – Registros

Para emular el aspecto que tendrían la tabla de registros y las gráficas de la página de listados ante un registro real, hemos creado una función que añade registros aleatorios al *array* enviado a la plantilla de listados.

Como hemos visto antes, el controlador manda un *array* de Registros a la plantilla para que este los muestre, Pues bien hemos creado la siguiente función para añadir más registros y así aumentar el número de muestras de las gráficas y el número de filas de la tabla.

```
public function crearRegistros($entries)
    {
        $i;
        for ($i=1; $i<300; $i++)
        {
                $user = rand(0,1);
                $entry = new \TfgBundle\Entity\Registro();
                if($user==0)
                {
                        $entry->setUsuario('eloy');
                        $entry->setIdUsuario(1);
                }else{
                        $entry->setUsuario('paco');
                        $entry->setIdUsuario(2);
                }
                $entry->setAccion('accion');
                $entry->setUrl('url');
                $dia = rand(1, 31);
                $mes = rand(7, 12);
                $fecha = new \DateTime('2015-'.$mes.'-'.$dia);
                $entry->setFecha($fecha);
                $entries[] = $entry;
        }
        return $entries;
    }
```

Además con esta función también nos haremos una idea del tiempo de ejecución para distinto número de registros, aunque únicamente en el momento de la muestra, no de la consulta.

4 – Otras pruebas

El resto de pruebas realizadas han estado más orientadas al procesamiento de formularios y al funcionamiento de las distintas rutas. Estas pruebas consistían simplemente en acceder a cierta ruta o en rellenar un formulario con datos incorrectos para comprobar la detección de errores. Entre estas también podemos destacar el uso de funciones para imprimir variables concretas como *echo* para cadenas y variables singulares y *print_r* para *arrays* y objetos.

7. Conclusiones y trabajo futuro

El objetivo básico de esta aplicación es la de dar información al administrador sobre los accesos de los usuarios a un servidor web, para así descubrir usos irregulares y actuar para evitarlos. De esta manera llegamos al objetivo principal, la seguridad.

Todas las mejores deberían ir orientadas en esa dirección, sin embargo, propondremos mejoras en todos los ámbitos posibles:

- La relación entre la clase Socio y las otras dos entidades está basada en los atributos "id" y "nombre", es decir, en la clase Registro hay un atributo de tipo *string* "usuario" y otro de tipo *integer*, "idUsuario". La mejora consistiría en que hubiera un solo atributo Socio, lo cual no hemos implementado debido a que resultaba costoso que por cada registro hubiera que buscar el objeto en la base de datos.

- En vez de realizar consultas al registro de acceso, una gran mejora sería configurar los registros de Symfony para que los registros no se escribieran en dicho fichero, sino en una tabla de la base de datos. Es posible que pudiera parecer costoso, pero a la hora de realizar consultas sería mucho más fácil y rápido, y además, habría facilitado mucho el desarrollo de esta aplicación y la mejora del punto anterior.

- Control de errores mediante instrucciones "try catch", pese a que Symfony ofrece mensajes de error en los formularios, así como una estructura bastante sólida a la hora de evitar errores de interacción con la base de datos, sería una buena idea incluirlas para mejorar la detección de errores.

- En términos de *framework*, Symfony podría ofrecer mayores facilidades a la hora de incluir librerías PHP de terceros. En un principio, las gráficas iban a ser implementadas mediante una de estas librerías, pero fue tan complicado incluirlas que vimos más sencillo realizarlas mediante JavaScript.

- Incluir búsquedas de formatos extraños en los registros, es decir, registros con una estructura diferente o en un orden irregular. Eso aumentaría el número de problemas identificables por el administrador.

8. Anexos

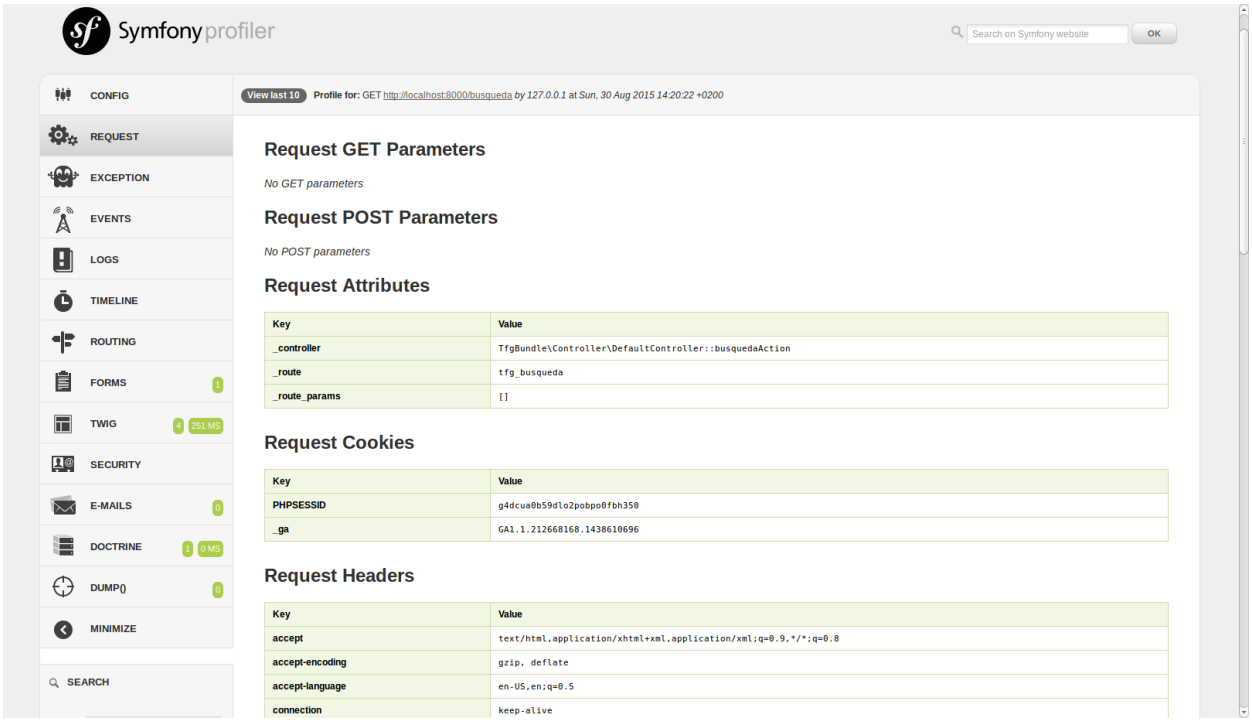
1 – Panel Symfony

Cuando accedemos a alguna de las rutas creadas, aparte de la página mostrada también veremos una especie de barra de herramientas en la parte de abajo, como si se tratara de un pie de página. En esta barra podremos ver información referente al proyecto Symfony, así como a la propia página. A continuación vemos un ejemplo:

Aquí vemos datos como la versión de Symfony (2.7.3), el controlador que ha procesado la petición y la acción ejecutada (DefaultController :: busquedaAction), formularios procesados, operaciones de bases de datos, coste temporal, etc.

Basta con pasar el puntero por encima para obtener más información sobre cada uno de los datos, pero si queremos obtener una visión mucho más completa accederemos al panel de Symfony haciendo clic en el apartado de configuración, el que va acompañado de engranajes:

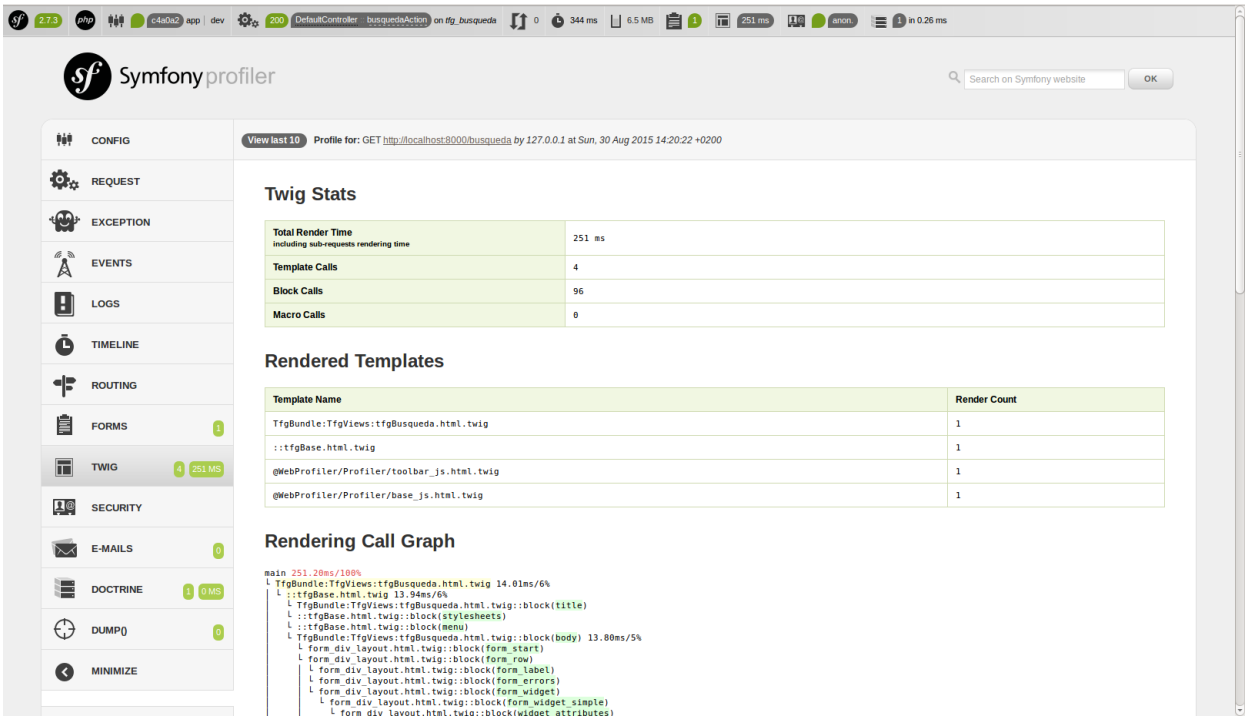
Una vez hagamos click nos saldrá esta página:



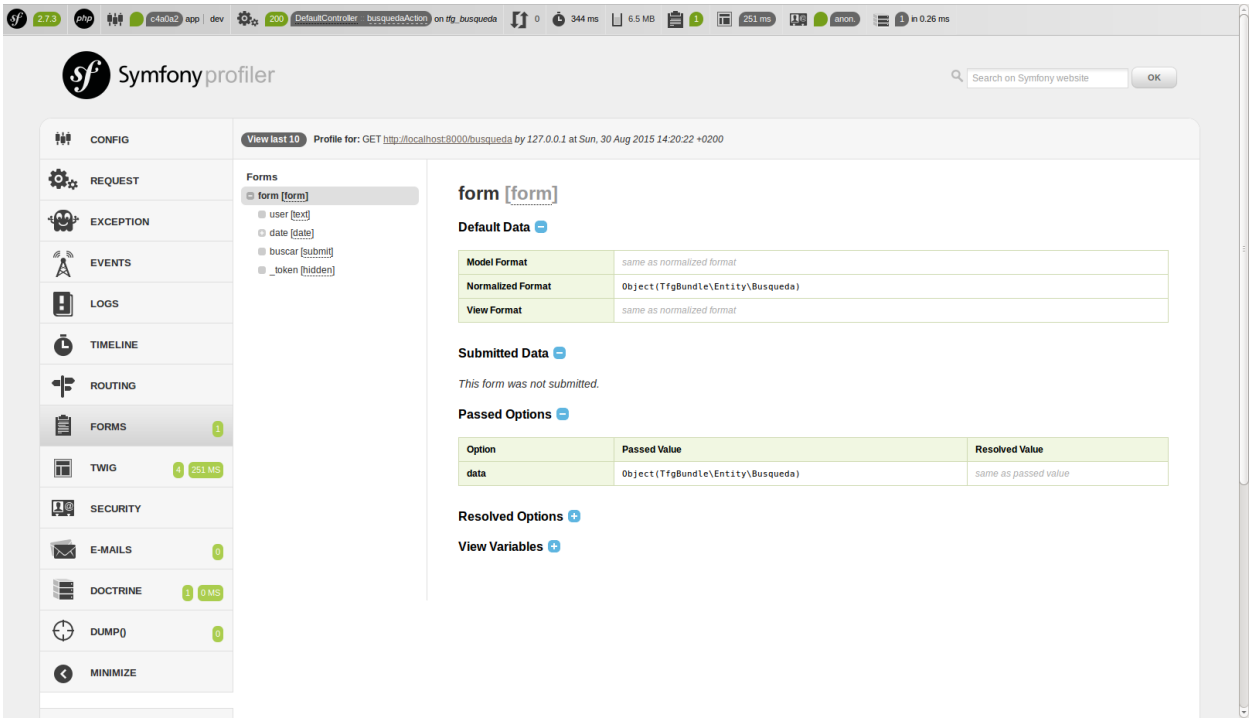
Como vemos aquí dispondremos de toda la información que podamos necesitar, dándonos la oportunidad de identificar incongruencias en la ejecución, como eventos inesperados o acciones no ejecutadas.

Además veremos la cantidad de plantillas cargadas y su ubicación en la sección correspondiente, así como información detallada sobre los formularios procesados:

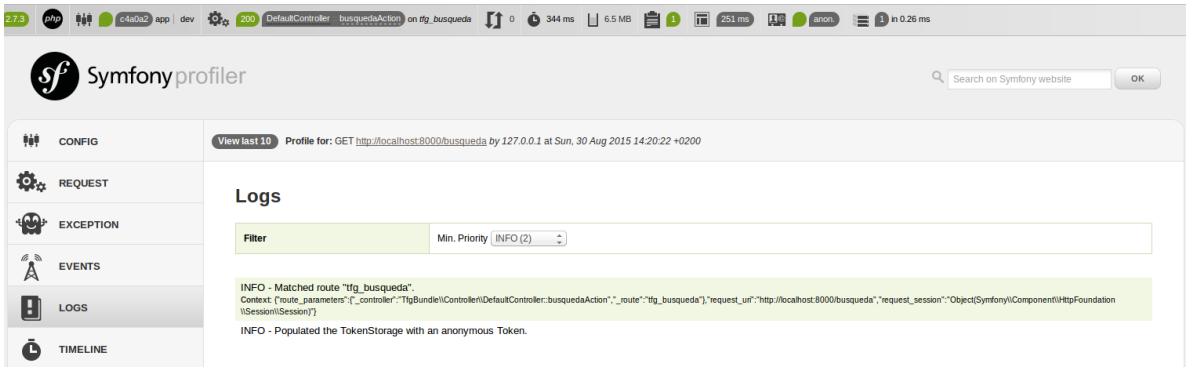
TWIG



FORMS



Otro dato interesante es que podremos ver la entrada generada al acceder a la ruta, nos referimos a la misma ruta que luego leeremos y mostraremos al usuario. Para ello iremos a la sección de LOGS:



2 – Inclusión de librerías

Hoy en día a la hora de desarrollar casi cualquier aplicación es necesario incluir librerías para facilitar el desarrollo y acceder a funcionalidades ya creadas. Pese a que Symfony contiene una gran cantidad de estas librerías hay algunas que tendremos que introducir nosotros.

Para explicar cómo se introducen vamos a poner como ejemplo Bootstrap, uno de los *frameworks* para desarrollo de proyectos de diseño más utilizados hoy en día. Para incluirlo lo primero que tendremos que hacer es colocarlo en la carpeta "public" del directorio "TfgBundle/Resources/"

Podemos copiar los archivos sueltos en la carpeta, aunque lo más correcto es crear una carpeta para cada librería, por tanto nosotros lo copiaremos en "public/css/bootstrap"

Una vez hecho usaremos la consola para situarnos en la carpeta del proyecto y a continuación ejecutaremos el siguiente comando:

```
php app/console assets:install
```

Este comando leerá esa carpeta, junto a otras tantas del proyecto, e incluirá su contenido en las librerías del proyecto para poder ser utilizado, más concretamente en el directorio "web/bundles/tfg", dependiendo esta última carpeta del nombre del *bundle*.

Ahora lo único que queda es insertarlo en la plantilla para que esta lo lea:

```
{% block stylesheets %}
        <link href="{{ asset('bundles/tfg/css/bootstrap/bootstrap.css') }}" type="text/css" rel="stylesheet" />
        <link href="{{ asset('bundles/tfg/css/tfg.css') }}" type="text/css" rel="stylesheet" />
{% endblock %}
```

Aparte de una librería ya hecha, también podemos usar este método para nuestras propias librerías, hojas de estilo y ficheros JavaScript, no obstante, si lo modificamos desde la carpeta "public", los cambios no tendrán efecto si no volvemos a ejecutar el comando anterior.

Este también será el método que utilizaremos para instalar la librería jqPlot, un *plugin* de jQuery dedicado a la creación de gráficas.

3 – Formularios

Symfony dispone de una herramienta para la creación fácil y sencilla de formularios, una especie de *helper* que nos permite especificar los campos, la tipología y las restricciones del mismo desde el controlador.

Una característica bastante interesante es que los formularios usan un objeto dado como modelo para rellenar el formulario, en otras palabras, puedes crear un objeto vacío y pasárselo al creador de formularios, cuando el usuario lo rellene y envié, estará rellenando sin saberlo los atributos del objeto. Aquí podemos ver un ejemplo de cómo funciona:

```
$busqueda = new Busqueda();
$busqueda->setDate(null);
$busqueda->setUser('all');

$form = $this->createFormBuilder($busqueda)
        ->add('user', 'text', array(
              'label' => false,
              'required' => false,
        ))
        ->add('date', 'date',array(
              'label' => false,
              'years' => range(2000, (int)date('Y')),
              'empty_value' => array(
                 'year' => 'Año',
                 'month' => 'Mes',
                 'day' => 'Día'
                 ),
              'required' => false,
        ))
        ->add('buscar', 'submit')
        ->getForm();

$form->handleRequest($request);
```

Podemos ver cómo hemos creado un objeto de clase Búsqueda y lo hemos usado como parámetro para crear el formulario. Asimismo vemos cómo se definen los campos y alguna de las opciones que podemos configurar como la visibilidad del título del campo, el rango de valores posibles o el valor que queremos por defecto.

En el ejemplo no lo vemos, pero también podemos poner una serie de restricciones que serán controladas automáticamente por Symfony, es decir, si ponemos un campo obligatorio y este no se rellena, se mostrará un mensaje de error antes de enviar el formulario. Lo mismo pasa con valores numéricos, a los cuales puedes poner valores máximos y mínimos o con las cadenas, cuya longitud podemos controlar.

Para que el formulario se muestre en la plantilla basta con añadir la siguiente línea en el fichero TWIG:

```
{{ form }}
```

No obstante, de esta forma, nos mostrará el formulario de forma predeterminada, la cual puede que no tenga un aspecto muy atractivo. En la plantilla podremos configurar, en cierta manera, la estructura que queremos para nuestro formulario para así mejorar la experiencia del usuario. Aquí mostramos un ejemplo:

```
{{ form_start(form) }}
    {{ form_errors(form) }}

    {{ form_row(form.user) }}
    {{ form_row(form.date) }}

    {{ form_rest(form) }}

    <input type="submit" />
{{ form_end(form) }}
```

Se puede ver que separamos los distintos elementos del formulario, teniendo un cierto control sobre el lugar donde mostramos cada elemento podemos modificar el formulario como más nos convenga.

A la hora de procesar el formulario lo haremos desde la misma función que lo crea mediante la función *isValid()*. Esta función devolverá *true* si el formulario se ha enviado, por tanto lo utilizaremos de la siguiente manera:

```
$form->handleRequest($request);

if($form->isValid())
{
        $em = $this->getDoctrine()->getManager();
        $em->persist($busqueda);
        $em->flush();
        …
```

Aquí vemos cómo podemos actuar sobre el objeto creado directamente, sin necesidad de completarlo de ninguna forma y, por poner un ejemplo, una aplicación dedicada en profundidad a la persistencia de objetos no necesitaría de muchas líneas de código para realizar su función, pues la mitad del trabajo la hará el creador de formularios.

4 – jqPlot

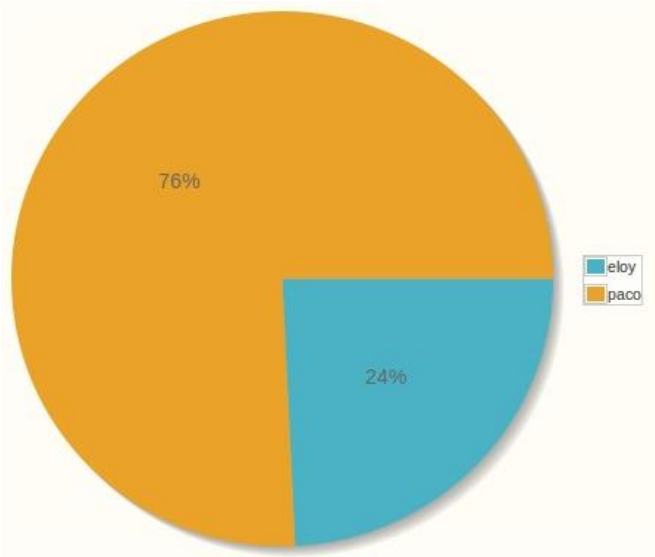
Puesto que esta es la librería menos conocida de las usadas, procederemos a explicar brevemente el uso que le hemos dado.

Como hemos dicho antes jqPlot es un *plugin* de jQuery orientado a la creación de gráficas, el cual hemos usado para obtener una visión más estadística del porcentaje de entradas de cada socio o del número de registros en cada fecha.

Para crear el gráfico circular basta con introducir las siguientes líneas en la plantilla:

```
<script>
    $(document).ready(function(){
        datos = [[]];
        {% for dato in datos %}
            datos[0].push(["{{ dato.user }}", {{ dato.cuenta }}]);
        {% endfor %}
        var plot1 = $.jqplot('socioEst', datos, {
            gridPadding: {top:0, bottom:38, left:0, right:0},
            seriesDefaults:{
                renderer:$.jqplot.PieRenderer,
                trendline:{ show:true },
                rendererOptions: { padding: 8, showDataLabels: true }
            },
            legend:{
                show:true,
                placement: 'inside',
                location:'e',
                marginTop: '15px'
            }
        });
</script>
…
…
<div id="socioEst">
</div>
```

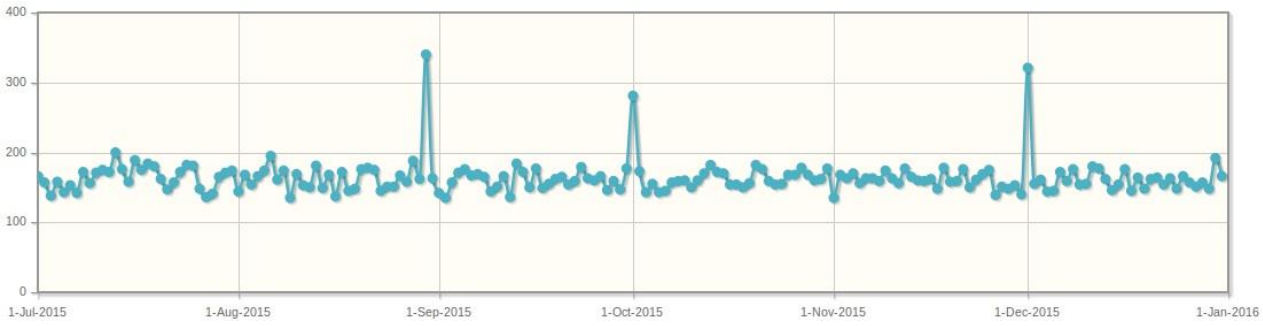
Como se puede apreciar, usamos una instrucción de flujo para rellenar una *array* que luego tomará la clase jqplot. El gráfico se declara como si fuera de cualquier tipo, pero donde se controla el tipo es en la variable *renderer*, que aquí toma el valor ":$.jqplot.PieRenderer", es decir, gráfico de tarta o circular. Dentro de la declaración del gráfico también podremos decidir la visibilidad y la posición. Aunque el tamaño del recuadro lo podemos controlar también aquí, usaremos hojas de estilo para hacerlo de un modo más intuitivo. Aquí vemos el resultado:



Para realizar la otra gráfica el proceso es muy similar, solo que usaremos otro *Renderer* para definir una gráfica por fechas ":$.jqplot.DateAxisRenderer":

```
$.jqplot('fechaEst',  datosFecha,
{
   axes:{
      xaxis:{
         renderer:$.jqplot.DateAxisRenderer,
         tickInterval:'1 month'
      },
      yaxis:{
         min:0
      }
   },
   seriesDefaults: {
      rendererOptions:{
         smooth: true
      }
   }
}
);
```

En este caso nos centramos en poner un valor mínimo al valor del eje y, para que no incluya valores negativos. Además definiremos el intervalo en que se mostrará el *label* del eje x dejando el aspecto final así:



9. Bibliografía

[1] http://es.wikipedia.org

[2] http://symfony.es

[3] Libro Desarrollo web ágil con Symfony2 de Javier Eguiluz.

[4] http://symfony.com

[5] http://librosweb.es

[6] http://thelozu.blogspot.com.es – Modelo Vista Controlador.

[7] http://dejensever.com – "Symfony2: Crear un bundle" de Amador López Parra.

[8] http://es.slideshare.net - Presentación dada para el DrupalCamp Guatemala 2013 – "Mis primeros pasos con Symfony 2" de Edgar Dueñas.

[9] Proyecto Final de Carrera – "Aplicación Web de bases de datos usando el framework Symfony" de Ángel Talavera Moreno